DOCUMENT RESUME

ED 349 938 IR 015 669

AUTHOR	Yoder, Sharon
TITLE	HyperTalk for Educators. An Introduction.
INSTITUTION	International Society for Technology in Education, Eugene, OR.
REPORT NO	ISBN-0-924667-95-8
PUB DATE	92
NOTE	291p.
AVAILABLE FROM	International Society for Technology in Education, 1787 Agate Street, Eugene, OR 97403-1923.
PUB TYPE	Guides - Classroom Use - Teaching Guides (For Teacher) (052)
EDRS PRICE	MF01 Plus Postage. PC Not Available from EDRS.
DESCRIPTORS	Authoring Aids (Programing); *Computer Software Development; Elementary Secondary Education; *Hypermedia; Microcomputers; Programing Languages
IDENTIFIERS	Apple Macintosh; *HyperCard; *HyperTalk

ABSTRACT

This guide is designed to introduce teachers and students to the use of the HyperTalk programming language in order to extend the power of HyperCard and to suggest ways to use HyperCard in conjunction with HyperTalk in an educational setting. It is noted that previous familiarity with the Macintosh interface, Home stack, and HyperCard at the authoring level are necessary for using this guide. Chapters 0-2 provide a brief overview of HyperCard and simple material that is often covered in beginning books or workshops. Chapters 3 through 20 cover many of the important, but relatively easy-to-use ideas in Hypertalk programming, as well as an introduction to some important ideas in computer science. Chapters 21-25 introduce important ideas in programming and computer science: conditionals, looping, scope of variables, and parameter passing. The book is designed for use while working at the computer, and it is noted that the final chapters would best be used in conjunction with HyperCard projects or other sources of information about HyperCard. Individual chapters contain summaries of new HyperTalk words, keyboard shortcuts, and information on debugging and testing. The index includes references to all important concepts and to the first use of each new HyperTalk word. Three appendices present the keyboard shortcuts given in the book by category, a list of HyperTalk words by category, and an alphabetical listing of all HyperTalk words. (ALF)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

ED349938

U.S. DEPARTMENT OF EDUCATION
Office of Educational Research and Improvement
EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)

☐ This document has been reproduced as received from the person or organization originating it.

☐ Minor changes have been made to improve reproduction quality.

• Points of view or opinions stated in this document do not necessarily represent official OERI position or policy.

IR015669

BEST COPY AVAILABLE

"PERMISSION TO REPRODUCE THIS MATERIAL IN OTHER THAN PAPER COPY HAS BEEN GRANTED BY
D. Moursund

TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)."

About the Authors

Sharon Yoder has taught mathematics and computer science at the junior high and high school level for 15 years. Her most recent public school experience was as a secondary computer science teacher and a computer coordinator involved in developing system-wide computer curriculum and in planning teacher inservice training.

In addition, she has taught mathematics, computer science, and computer education at a number of universities in northeastern Ohio, including Kent State University, the University of Akron, and Cleveland State University. She worked closely with the College of Education of Cleveland State University in developing their computer programming courses for teachers.

After a year as an Education Specialist for Logo Computer Systems, Inc., Sharon has returned to teaching. She is currently at the University of Oregon, where she teaches computer education courses.

For the past several years, she has conducted workshops and presented papers at conferences nation-wide and has been involved in a number of book publishing projects, including the *Nudges* series, *Introduction to Programming in Logo Using LogoWriter*, and *Introduction to Programming in Logo Using Logo PLUS*. In addition, she has been a frequent contributor to *The Computing Teacher*. She is currently editor of the *Logo Exchange* and *Microsoft Works in Education* newsletter.

Der-Thanq (Victor) Chen is currently a doctoral candidate in Computer in Education at the University of Oregon. His major areas of interest include hypermedia and teacher training. He has served as a graduate teaching assistant and has taught the hypermedia course for students obtaining masters degrees in Computers in Education.

David John Burrowes majored in cognitive science at Brown University. He has served as a technical consultant on this project as well as others in the areas of Logo and telecommunications. David is currently employed as a programmer by Blyth Software.

Editor: Ron Renschler

Production: Sharon Yoder

©International Society for Technology in Education, 1992
1787 Agate Street, Eugene, Oregon 97403-1923
Orders: 800/336-5191
Phone: 503/346-4414
Fax: 503/346-5890
CompuServe: 70014,2117
Internet: ISTE@Oregon.uoregon.edu

ISBN 0-924667-95-8

HyperTalk for Educators

An Introduction

by Sharon Yoder

with

David John Burrowes

and

Der-Thanq (Victor) Chen

© International Society for Technology in Education, 1992
ISBN 0-924667-95-8

Contents

Introduction 1

What Do You Need to Know? 1
What Hardware and Software Do You Need? 2
Who Should Use This Book? 2
The Structure of This Book 3
How to Use This Book 3

Chapter 0. A Brief Review 5

Getting Started 5
What Is *HyperCard*? 5
What You Need to Know to Begin 9
A Brief Review—Creating a Stack 9
A Brief Review—Creating a New Card 12
A Brief Review—Editing a Background 14
A Brief Review—Creating a New Background 16
A Brief Review—Creating a Button 17
A Brief Review—Creating a Field 20
Keyboard Shortcuts 23

Chapter 1. Using Visual Effects 25

Introduction to Scripting 25
Visual Effects 29
Debugging and Testing 32
New HyperTalk Words in This Chapter 33
Project Ideas 34

Chapter 2. Boxes, Windows, and More 35

Windows 37
Show and Hide 38
More on Show and Hide 40
The Message Box as a Container 41
Functions in Scripts 43
Debugging and Testing 44
New HyperTalk Words in This Chapter 44

Chapter 3. Working With Fields 47

Using Buttons to Work With Fields 50
Scripting Fields 51
Using Number 55
Debugging and Testing 58
New HyperTalk Words in This Chapter 59
Project Ideas 60

Chapter 4. Introduction to Button Properties 61

Getting Started Using HyperTalk With Button Properties 61
Using Set 63
Yet More Properties 66
Debugging and Testing 69
New HyperTalk Words in This Chapter 71
Project Ideas 71

Chapter 5. The Play Command 73

Representing Notes 73
Using the Play Command 75
Debugging and Testing 76
New HyperTalk Words in This Chapter 77
Project Ideas 77

Chapter 6. Using Field Properties 79

Getting Started With Field Properties 79
Working With Field Properties 82
More With Field Properties 84
Debugging and Testing 86
New HyperTalk Words in This Chapter 87
Project Ideas 87

Chapter 7. Overview of HyperTalk Grammar 89

Commands and Functions 89
Syntax Descriptions 91
Objects 94
Properties 94
Values 95
Messages 95
Scripts 100
Debugging and Testing 101
New HyperTalk Words in This Chapter 102
Project Ideas 102

Chapter 8. More About Properties 105

Global Properties 105
Window Properties 108
Debugging and Testing 110
New HyperTalk Words in This Chapter 111
Project Ideas 111

Chapter 9. Design Issues 113

Stack Structure 113
Using Visual Effects 117
Backgrounds 119
Conventions for Using Icons 120
Some Other Considerations 123
Debugging and Testing 124
Project Ideas 125

Chapter 10. More on Button and Field Properties 127

Using Visible 127
Working With Fonts in Fields 129
Working With Fonts on Buttons 130
Debugging and Testing 134
New HyperTalk Words in This Chapter 135
Project Ideas 135

Chapter 11. Writing Your Own Message Handlers 137

Making HyperTalk Easier to Read 137
Using Handlers More Than Once 139
Testing Your Handlers 140
Debugging and Testing 141
New HyperTalk Words in This Chapter 142
Project Ideas 142

Chapter 12. Object Descriptors, Constants, Ordinals, The, and Me 143

Object Descriptors 143
Special Object Descriptors 143
Constants 146
Special Ordinals 147
Using The 148
Debugging and Testing 148
New HyperTalk Words in This Chapter 149
Project Ideas 149

Chapter 13. Working With the Tool Window 151

Using Tools 151
Clicking and Dragging 152
Painting Properties 156
Debugging and Testing 160
New HyperTalk Words in This Chapter 163
Project Ideas 164

Chapter 14. Printing 165

Printing Stacks 165
Printing Reports 167
Printing Cards 168
Printing Scripts 169
New HyperTalk Words in This Chapter 171
Project Ideas 171

Chapter 15. Animation 173

Flipping Cards 173
Using the Tool Window 175
Debugging and Testing 178
New HyperTalk Words in This Chapter 178
Project Ideas 179

Chapter 16. DoMenu....181

Debugging and Testing....185
New HyperTalk Words in This Chapter....185
Project Ideas....186

Chapter 17. Message Passing....187

Debugging and Testing....191
New HyperTalk Words in This Chapter....192
Project Ideas....192

Chapter 18. Cards, Backgrounds and Stacks–Properties and Scripts....193

Card and Background Properties....193
Stack Properties....195
Scripting Cards, Backgrounds, and Stacks....195
Placing Scripts....197
Debugging and Testing....198
New HyperTalk Words in This Chapter....199
Project Ideas....199

Chapter 19. Working With Text....201

Chunks and Chunk Expressions....202
Some Functions and Commands for Working With Text....203
Some Constants and Operators for Working With Text....208
A Note on HyperText....212
Debugging and Testing....212
New HyperTalk Words in This Chapter....213
Project Ideas....213

Chapter 20. Interactive Programming....215

Ask and Answer....215
Debugging and Testing....221
New HyperTalk Words in This Chapter....221
Project Ideas....221

Chapter 21. Conditionals....223

Single Statement If-Then....223
Multiple Statement If-Then....224
Single Statement If-Then-Else....225
Multiple Statement If-Then-Else....225
Using the If Structure....226
Debugging and Testing....231
Project Ideas....233

Chapter 22. Math Functions....235

Common Math Functions....235
Mathematical Functions....237
New HyperTalk Words in This Chapter....238
Project Ideas....238

Chapter 23. Containers....239

Variables....239
Using Variables....241
More About It....243
Using Global....244
Debugging and Testing....245
New HyperTalk Words in This Chapter....246
Project Ideas....246

Chapter 24. Looping....247

Repeat For....247
Repeat With....248
Repeat While and Repeat Until....250
Modularizing Your Scripts....252
Wait Loops....252
Debugging and Testing....253
New HyperTalk Words in This Chapter....254
Project Ideas....254

Chapter 25. Using Parameters....255

Getting Started With Parameters....255
More Than One Parameter....258
Debugging and Testing....260
Project Ideas....261

Appendix 1. Keyboard Shortcuts....265

Appendix 2. HyperTalk Words by Category....269

Appendix 3. Alphabetical Listing of HyperTalk Words....279

Index....285

Introduction

HyperCard, a software package that runs on Macintosh computers, was introduced in 1987. Since that time, *HyperCard* has become increasingly popular. More recently, various *HyperCard*-like products have been introduced both for the Macintosh and for MS-DOS computers. Now, both teachers and students are using these new products in the classroom.

While *HyperCard* and its competitors are new to educational computing, the underlying concepts behind these products are not new. All of these products allow you to produce nonsequential documents. The idea of nonsequential text has been talked about for many years. The term to describe such documents, hypertext, was coined by Ted Nelson in his book Computer Lib/Dream Machines, published in 1974. A familiar example of a noncomputer hypertext document is a "choose your own adventure" book. Using *HyperCard* you can use a Macintosh computer to produce your own hypertext documents. With *HyperCard* you can easily add graphics and sound to your documents; you can also access videodiscs and other external media. The combination of text and other media into nonsequential documents produces hypermedia documents. *HyperCard* makes creating such documents quite easy.

If you have experience with the Macintosh, you will have little difficulty getting started with *HyperCard*. You can produce quite sophisticated hypermedia documents using the pull-down menus that appear at the top of the screen when you start *HyperCard*.

In addition to the many menu-based features, *HyperCard* has a programming language called **HyperTalk** built into it. This is a special programming language designed specifically for working in the *HyperCard* environment. This book is designed to introduce teachers and students to the use of the HyperTalk programming language. Its purpose is to present the elements of HyperTalk as a programming language that can be used to extend the power of *HyperCard*. It also provides suggestions of ways in which you can use *HyperCard* in conjunction with HyperTalk in an educational setting.

A majority of the books about *HyperCard* and HyperTalk on the market today are either very thick (and overwhelming) or rather thin and dense. Books in the former category usually present everything the reader should do, mouse click by mouse click. This is very useful for novice users and gives them a feeling of confidence as they begin. Books in the latter category often take the form of a reference work. Such books usually provide an overview of *HyperCard*; give technical, syntactic descriptions of HyperTalk commands; and include only a few brief examples.

HyperTalk for Educators—An Introduction is designed to fall between the above two categories. It does not provide a mouse-click-by-mouse-click explanation for every example. Nor does it attempt to include everything that you can do with *HyperCard*. Rather, this book leads you through some simple examples that demonstrate each new idea as it is presented. It provides suggestions of activities you can do to practice new ideas. It connects HyperTalk to computer science concepts and other programming languages.

What Do You Need to Know?

Although the book begins with a brief review of *HyperCard* objects, it is *not* truly a beginning book. To use this book you should:

- be familiar with the Macintosh interface
- have explored the Home stack and have become familiar with its contents
- have used *HyperCard* at the "authoring" user level; that is, you should have explored some *HyperCard* stacks and created some cards, buttons, and fields
- have begun to want to learn about HyperTalk in order to use *HyperCard* more extensively than is possible at the authoring level.

If you have not worked with *HyperCard*, you should spend some time with a beginning book. A good choice would be *HyperCard for Educators—An Introduction* by Bull and Harris. This book was designed for and tested with educators and, along with some practice with *HyperCard*, should serve as excellent preparation for this book.

Readers who have both extensive experience with the Macintosh and know one or more programming languages may find that *HyperTalk for Educators* alone will meet their needs. However, even such experienced computer users should have a reference available to provide the background not discussed in detail in this book. An excellent source for the more experienced user is *Apple HyperCard Script Language Guide: The HyperTalk Language*. Although this book is about HyperTalk, it does provide an overview of *HyperCard* itself and will serve as a technical reference for more advanced ideas. Currently, this book is only available for *HyperCard* Version 1.2.5. A more advanced user may also wish to acquire a book such as *HyperTalk 2.0: The Book*, by Winkler and Kammins, which examines the features of *HyperCard* Version 2.0.

What Hardware and Software Do You Need?

To run the *HyperCard* program, it is best to have a Macintosh with at least one megabyte of memory. While it is possible to use *HyperCard* with two 800K floppy-disk drives, you can quickly find yourself doing a great deal of "disk swapping." Thus, it is best if you have a hard drive for your work with *HyperCard*.

Versions of *HyperCard* fall into two categories—those before Version 2.0, and Version 2.0 and after. You do not need to have *HyperCard* Version 2.0 to use this book. Nearly all the activities covered in this book will work exactly as described if you use Version 1.2.5. Features unique to Version 2.0 will always be noted. However, if you have *HyperCard* Version 1.0, 1.1, or 1.2, you will occasionally encounter features that have been added to Version 1.2.5 that will not be available in these earlier versions. It would be best if you acquired at least Version 1.2.5 to work with this book.

Throughout the book there are "technical notes" to provide details not directly related to the examples or ideas being presented. Differences between versions of *HyperCard* will often be given in these notes. All technical notes appear in this style:

Technical Note: This is an example of a technical note.

The pictures of screens in this book were created using Version 2.0. If you are using an earlier version of *HyperCard*, your screen will occasionally look somewhat different. In general, the different appearance of the screen will not affect the way examples work.

Who Should Use This Book?

This book can be used by individual teachers who are comfortable with exploring new software and who want to learn HyperTalk for their own purposes. It is designed for self-study, preferably where the learner has other references, whether printed, online, or human! The book could also be used in secondary school classes with students who want to learn to program in HyperTalk. The book can also be used for staff

development classes. As hypermedia environments become more and more common in our schools, increasing numbers of students will want to learn to use *HyperCard* to produce their own products. Increasing numbers of educators will want to know enough about *HyperCard* to help students produce hypermedia projects. This book can be used to help both teachers and students develop more advanced skills with *HyperCard*.

Although in recent years there has been a decrease in the teaching of traditional programming in schools, *HyperCard* is increasingly being taught at a variety of grade levels. Learning HyperTalk can be a natural extension to using *HyperCard* menus to create projects. By learning HyperTalk, students can learn some of the fundamentals of programming. It seems likely that many of the applications both students and adults will use in the future will include programming languages. Even now we have spreadsheet languages, database languages, and the ability to write macros for telecommunications programs. The sophisticated computer users of the future will most likely be more productive if they learn some elements of computer programming. Thus, this book could be used in a secondary classroom to help students learn to use HyperTalk with their *HyperCard* projects.

The Structure of This Book

Although the chapters in this book are numbered sequentially, the book really consists of three parts. The first part of the book is a review. Chapter 0, so named because the assumption is that the reader already knows what is covered in the chapter, provides a brief overview of *HyperCard*. The next couple of chapters provide simple material that is often covered in beginning books about *HyperCard* and in beginning workshops or training sessions.

Chapters 3 through 20 cover many of the important, but relatively easy-to-use ideas in HyperTalk programming. At the same time, these chapters provide an introduction to some important ideas in computer science. Working through these chapters in sequence, with time allotted to practice new ideas, should provide readers with a good understanding of the HyperTalk programming covered.

Chapters 21 through 25 introduce important ideas in programming and computer science: conditionals, looping, scope of variables, and parameter passing. The material in these last few chapters is more difficult for those who have had no previous experience in programming. These chapters would best be used in conjunction with *HyperCard* projects and/or other sources of information about *HyperCard*. There are many *HyperCard* books on the market that address *HyperCard* features and give the details of syntax for HyperTalk words not discussed in this book. However, few of these sources provide an overview of the programming concepts that are covered in this book. The novice programmer should spend considerable time creating new *HyperCard* projects, making use of other sources, and working through the last few chapters of this book.

How to Use This Book

This book is designed so that you can use it while working at the computer. You should read a section and immediately try the ideas discussed in that section. Take time to "play around" with new ideas. Don't simply duplicate the examples in the book. Extend them. Explore. Experiment. If you simply rush from example to example and from section to section, you will quickly become overwhelmed with new ideas and concepts.

Each chapter of the book has a summary of the new HyperTalk words in that chapter. When you get to the end of a chapter, take time to be sure you have worked

with each of the new words. Often the chapter will also contain a section giving keyboard shortcuts. These shortcuts should be reviewed from time to time since you will only remember them as you learn to make use of them. And don't ignore the section in many chapters on debugging and testing. These sections are designed to help you improve your ability to problem solve when working with *HyperCard*.

Finally, the book has a number of reference sources to help you find the information you need.

- The Table of Contents lists every major section in the book.
- The Index includes references to all important concepts as well as a reference to the first use of each new HyperTalk word.
- Appendix 1 presents by category all of the keyboard shortcuts given in the book.
- Appendix 2 lists all HyperTalk words by category (commands, functions, properties, and so forth.)
- Appendix 3 lists all HyperTalk words alphabetically. The category of each HyperTalk word is also given.
- The detailed syntax of each HyperTalk word is given in the text of the book. These can be located by using the index.

These sources can be used to find information when you don't remember exactly where in the text of the book it was introduced. Even after you have worked with HyperTalk for some time, you will find these references helpful. HyperTalk is such a large programming language that it is unlikely that you will be able to remember all of the commands and keystrokes you can use.

Chapter 0. A Brief Review

This chapter provides a brief review that describes how to use buttons, fields, cards, backgrounds, and stacks in *HyperCard*. It is designed for people who have had some experience with *HyperCard*. If you have never used *HyperCard* before, you should work with more complete introductory material before beginning this book.

Getting Started

This book assumes that you have had experience using the Macintosh series of computers and have had some experience with *HyperCard*. It assumes you have become familiar with some of the stacks that come with *HyperCard* and have created your own stacks, using cards, backgrounds, fields, and buttons. It does not teach everything you need to know to get started using *HyperCard*. If, as suggested in the Preface, you work through the book *HyperCard for Educators—An Introduction*, by Bull and Harris, and spend some time experimenting with *HyperCard*, then you will have sufficient background to begin using this book.

Perhaps you have worked with *HyperCard*, but some weeks or months have passed since that exposure. Perhaps you took a day-long workshop or worked with a friend on a *HyperCard* project. This initial chapter provides the kind of brief review you need to be ready for the material in Chapter 1 on scripting. If you currently feel quite comfortable using *HyperCard*'s pull-down menus, you should skip immediately to Chapter 1.

What Is *HyperCard*?

If you ask several people what *HyperCard* is, you are likely to get several different answers. Some people see *HyperCard* as an authoring system that can be used to write Computer Assisted Instruction or hypertext documents. Yet others think of *HyperCard* as a kind of extended database. Some think of *HyperCard* as a tool to control other media such as CD-ROMS and videodisc players. Still others think of *HyperCard* as a tool they can use to keep track of appointments, make schedules, and file their addresses. *HyperCard* has been described as an electronic doodling pad and as an electronic "flip book." It has been used to create interactive guides to buildings and conferences, electronic yearbooks, and programs to explore art or music.

Why is there such variety in the way people think about *HyperCard*? No doubt the differing viewpoints occur because *HyperCard* is such a flexible computer application. It does not fit nicely into our standard categorizations of tool-type software: word processors, databases, spreadsheets, telecommunications, and graphics packages. It is some of each of these and more.

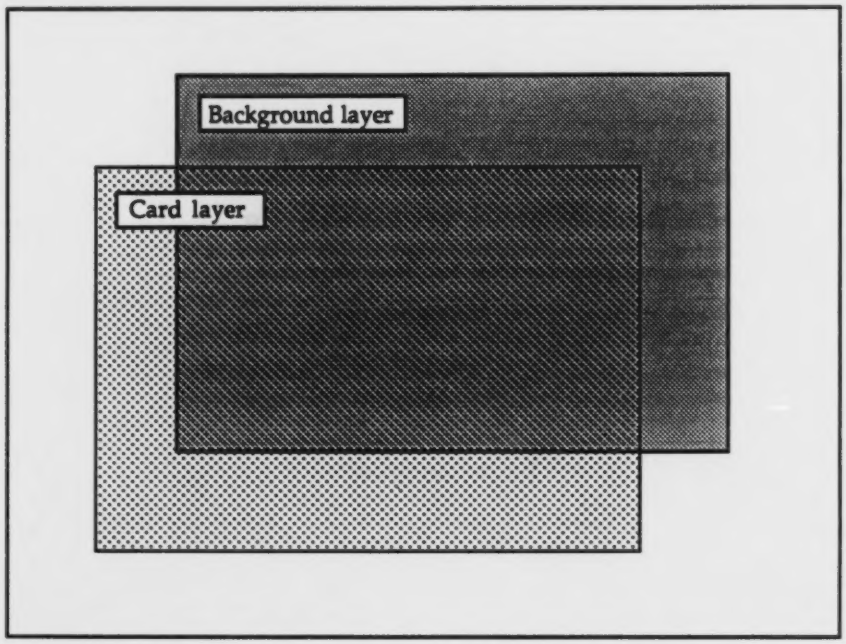
Indeed, then what is *HyperCard*? Step back for a moment and look at what components are available when working with *HyperCard*. Each application you use on a Macintosh produces some sort of document. A spreadsheet application produces a

spreadsheet document; a graphics application produces a picture document of some kind. In *HyperCard*, the documents are called **stacks**. Whenever you open a *HyperCard* document, you have opened a stack.

What is in a stack? The first and most obvious component of a stack is a **card**. You can think of a *HyperCard* stack as being much like a pile of index cards. Every *HyperCard* stack contains one or more of these index cards. Cards are the building blocks out of which you create stacks. Information in *HyperCard* is stored in the cards that make up a stack, just as information is put on index cards stored in a card file or recipe box.

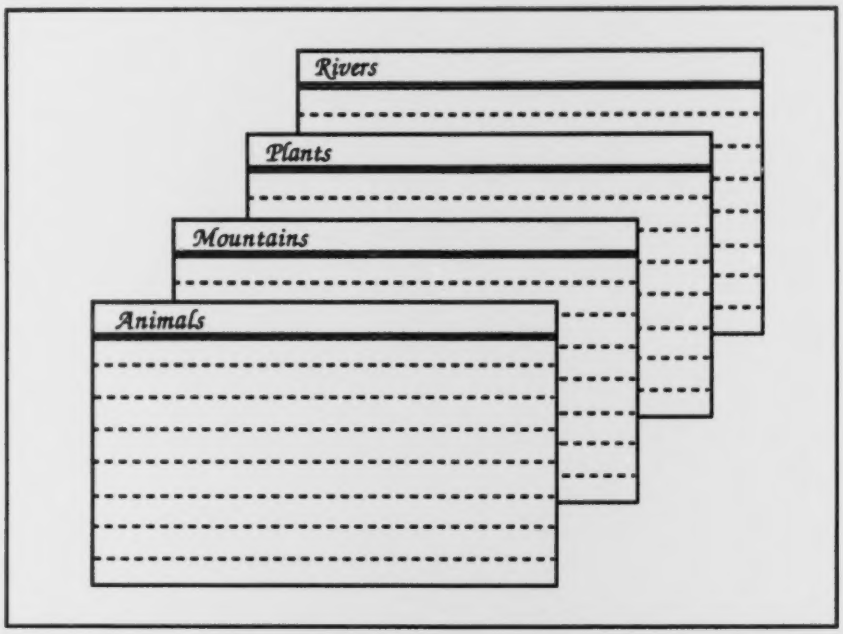
To make this model of *HyperCard* cards more accurate, think of a card as being made of clear plastic or glass. You can write directly on the card (glass) and you can stick things on the card, but at the same time you can see through the card.

Why should you think of the card as transparent? Because each card in *HyperCard* has a **background**. You can see the background through the card, so whatever is put on the background is visible through the card (unless you put something onto the card on top of an object on the background):



Yet another way to think about cards and backgrounds is to think of something familiar, like the kind of index cards that have been preprinted. The index cards you buy are all alike—often they have a double pink line at the top and light blue lines on the rest of the card. Your stack of brand-new cards all have the same background. When you use your index cards, you write different things on each card. Although the

background of the cards (the printed lines) remains the same, each card looks different because you have put writing on the card that is on top of the predrawn background lines:



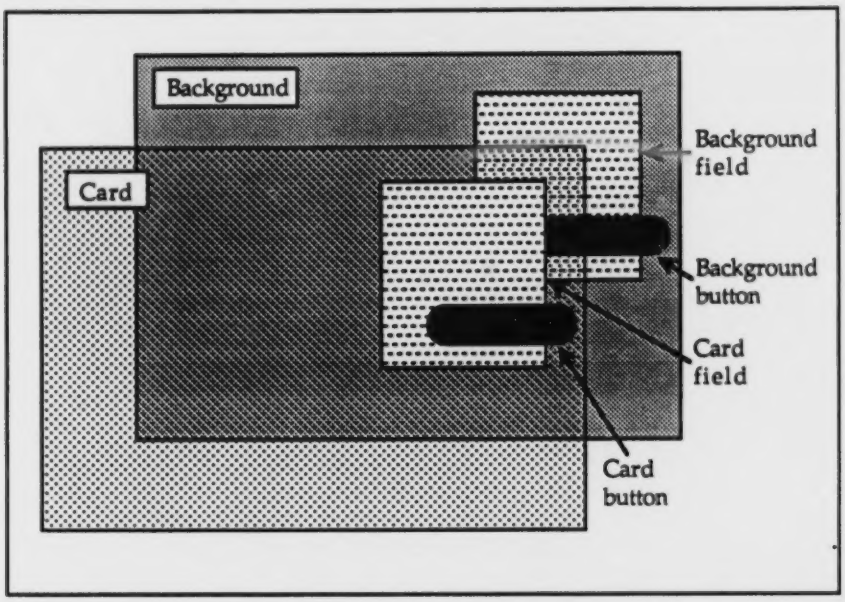
In most cases, a *HyperCard* stack has groups of cards having the same background, just like the index cards shown in the previous illustration. In *HyperCard* you write different things on each card in much the same way that you write different things on each index card. If you are working with a lot of index cards, you might use blue cards for one purpose, pink for another, and white for yet another. In *HyperCard* you use different backgrounds for different purposes.

Simple *HyperCard* stacks often have the same background for every card. In more complex stacks, however, the background may be used to distinguish between distinctly different parts of the stack. Learning to use backgrounds well is part of learning to create stacks that are easy to use.

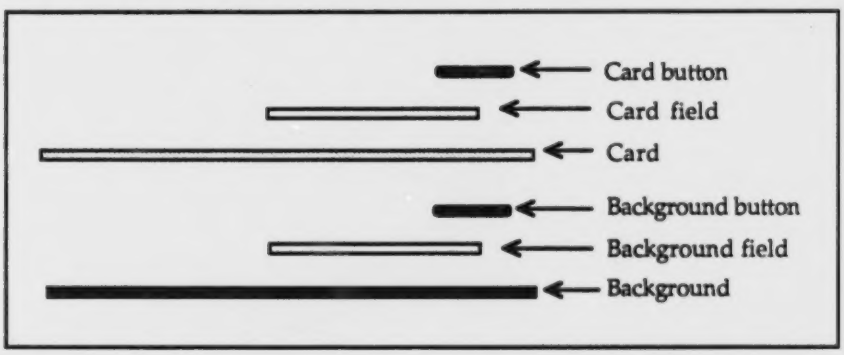
HyperCard provides tools for "painting" directly on both cards and backgrounds. These graphics tools allow you to create drawings, add borders, use patterns, and much more. In addition, you can add objects called buttons and fields both to backgrounds and cards. Thus you can have background fields and background buttons as well as card fields and card buttons. A more complete review of buttons and fields is given later in this chapter.

All the objects on a card or background are layered. That is, you can move some objects "on top of" other objects. If you create a single *HyperCard* card and make one

background field, one background button, one card field, and one card button *in that order*, then they are layered as shown in the following diagram:



The next diagram shows how the "layering" would appear if you were looking at the edge of the card with the parts separated:



Technical Note: The layering of buttons and fields is also dependent on the order in which they are created. The preceding diagrams are accurate only for the instance where a single card field is created, followed by a single card button, and for where a single background field is created followed by a single background button. If more than one button is created on the card, the last button created will be "above" the previously created button. However, card buttons and fields will always *be above background buttons and fields.*

In *HyperCard* stacks, backgrounds, cards, fields, and buttons are all called objects. Everything you create in *HyperCard* is done by using these objects; everything you see in stacks created by others is accomplished by using these objects. Sometimes you will hear people talking about *HyperCard* as an "object-oriented" programming environment. It would be more accurate to call *HyperCard* an "object-like" programming environment. True object-oriented languages allow you to create new types of objects; *HyperCard* allows you to create only the five kinds of objects previously described. While *HyperCard* retains some of the characteristics of an object-oriented environment, it does not have the full power of an object-oriented programming language.

What You Need to Know to Begin

This book assumes you have either worked with *HyperCard* in the "immediate mode" or you have programming experience and understand the Macintosh interface. That is, it assumes that you have either spent some time with *HyperCard* becoming familiar with the Home stack and have created stacks, cards, fields, and buttons, or that you know a good deal about programming languages and using the Macintosh.

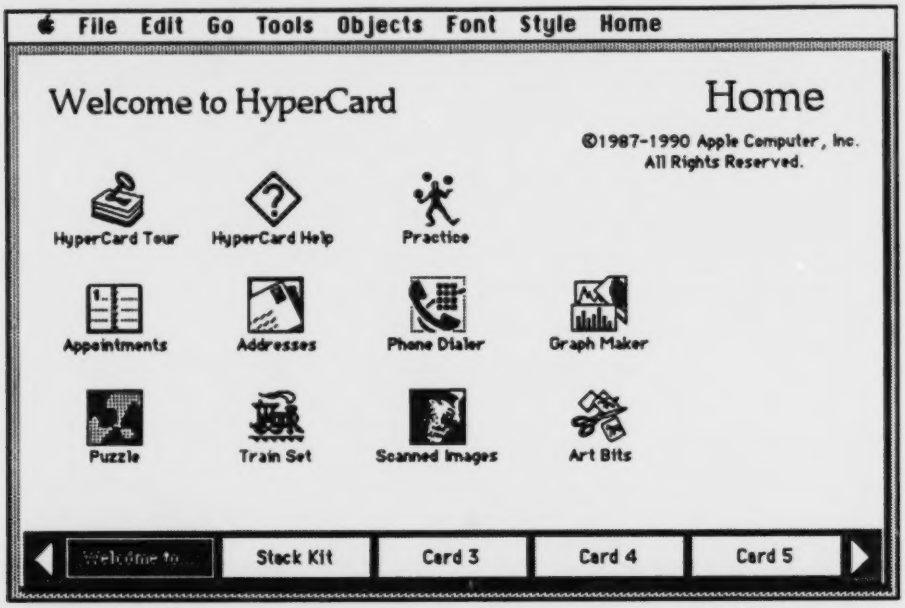
The ideas you should know to continue with this book are reviewed briefly in the following sections. If you find them difficult to follow, spend some time with a beginning *HyperCard* book or with the *HyperCard* Help stack until you are comfortable using pull-down menus and the mouse to create simple stacks containing buttons, fields, cards, and backgrounds.

A Brief Review—Creating a Stack

Recall that to start *HyperCard*, you can either click on the *HyperCard* icon and choose **Open** from the **File** menu or you can double click on the *HyperCard* icon. This starts *HyperCard* with the Home Card showing. If you are using Version 1.2.5, your screen should look something like this:

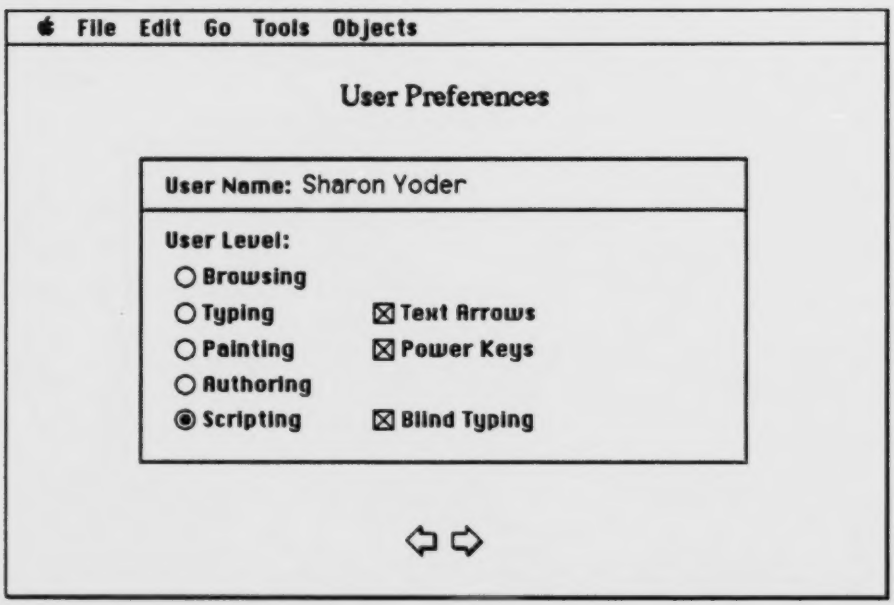
BEST COPY AVAILABLE

If you are using *HyperCard* Version 2.0, your screen should look something like this:



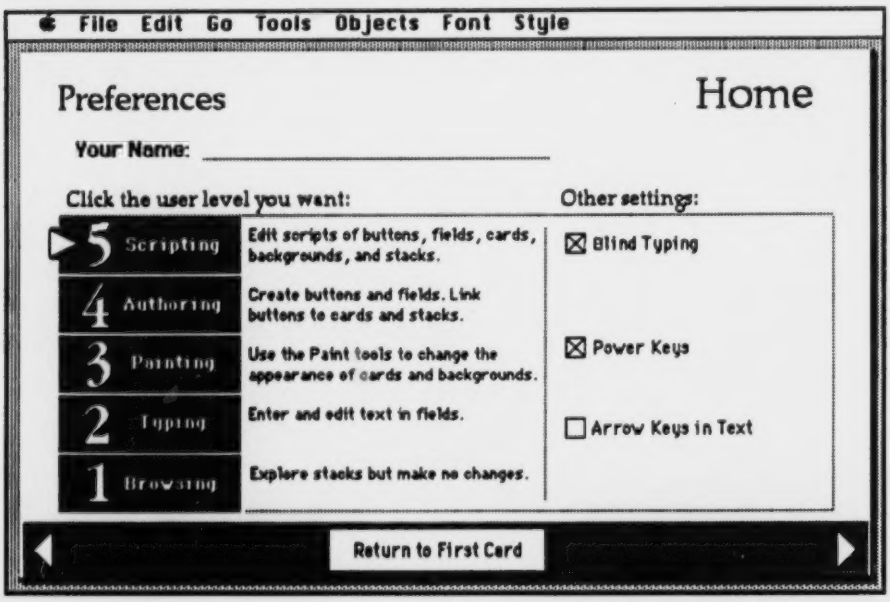
Technical Note: In this book, screens will be shown as they appear on the "standard" 9-inch Macintosh screen. That is, the Menu bar will appear along the top edge of the card. If you have a larger screen, the Menu bar will be above the card.

To begin a new *HyperCard* project, you first need to create a new stack. Before creating a new stack, however, you need to be sure that your User Level is set correctly. If you are using Version 1.2.5, click on the Left Arrow. You see the following card.



Click **Scripting** and then return to the Home Card by clicking on the Right Arrow. Note that the boxes on the right side of the Preferences card may be checked. These settings will be discussed later in the book.

If you are using Version 2.0, select **Prev** from the **Go** menu. Set the level to 5 by clicking on **Scripting**.


File Edit Go Tools Objects Font Style
Preferences
Home
Your Name:
Click the user level you want:
Other settings:
5 Scripting
Edit scripts of buttons, fields, cards, backgrounds, and stacks.
4 Authoring
Create buttons and fields. Link buttons to cards and stacks.
3 Painting
Use the Paint tools to change the appearance of cards and backgrounds.
2 Typing
Enter and edit text in fields.
1 Browsing
Explore stacks but make no changes.
Blind Typing
Power Keys
Arrow Keys in Text
Return to First Card

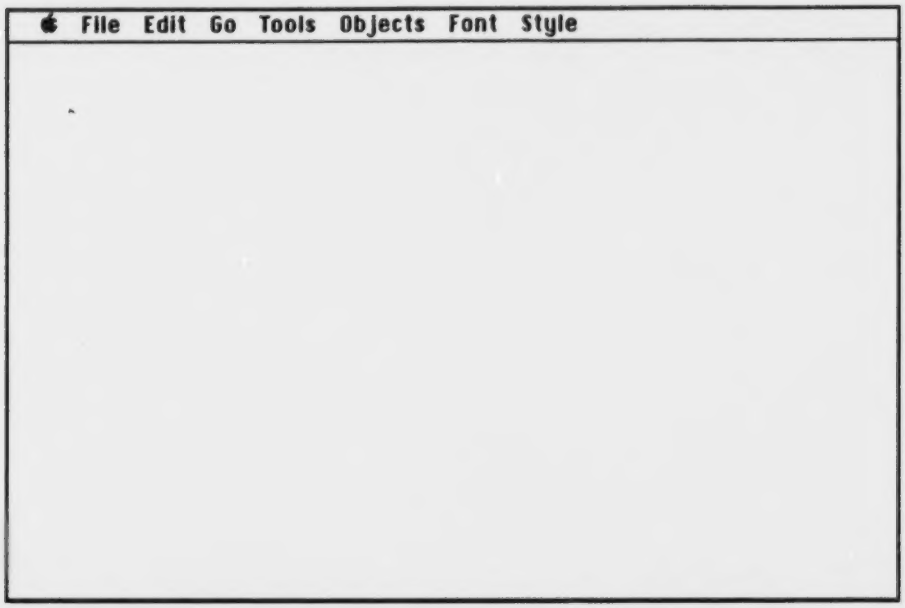
Then return to the Home Card by clicking on the Right Arrow. Note again that different boxes on the right side of the Preferences card may be checked.

*Technical Note: If **Painting, Authoring, and Scripting** do not appear, select **Message** from the **Go** menu. When the Message box appears, type the word "magic." Close the Message box by clicking in the small square in the upper left corner. Now you are ready to set the User Level. Note that you must be on the Preferences card for this to work correctly.*

To create a new stack, choose the **File** menu and select **New Stack**. You will then be asked to name your stack. Below the box where you type the name of your new stack, there is a check box that allows you to decide whether to use the current background in your new stack. In Version 1.2.5, you must *remove* the check from that box by clicking on the "X." Name your stack and click on **New**. You then have a new stack that contains a single card with a blank background.

Technical Note: If the box is checked, this will cause HyperCard *to copy the background and stack from the "current" stack and use them to build the new stack. This means that your new stack has all the scripts that were in both the background and the stack of the current stack. You also get copies of all of the resources that were in the original stack.*

Your new stack will look like this:



Keep in mind that you are looking through an entirely blank, transparent card at a completely blank background.

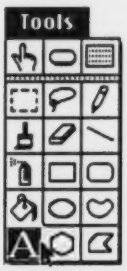
A Brief Review—Creating a New Card

Your stack already contains one card. You can create a new card by choosing **New Card** from the **Edit** menu. (If you prefer, you can press Command-N to create a new card.) Nothing on your screen changes. How do you know that you actually created a new card? Go to the **Objects** menu and choose **Card Info**. A dialog box appears that gives you information about your card. Most likely it says that you are on card 2 of 2 cards.

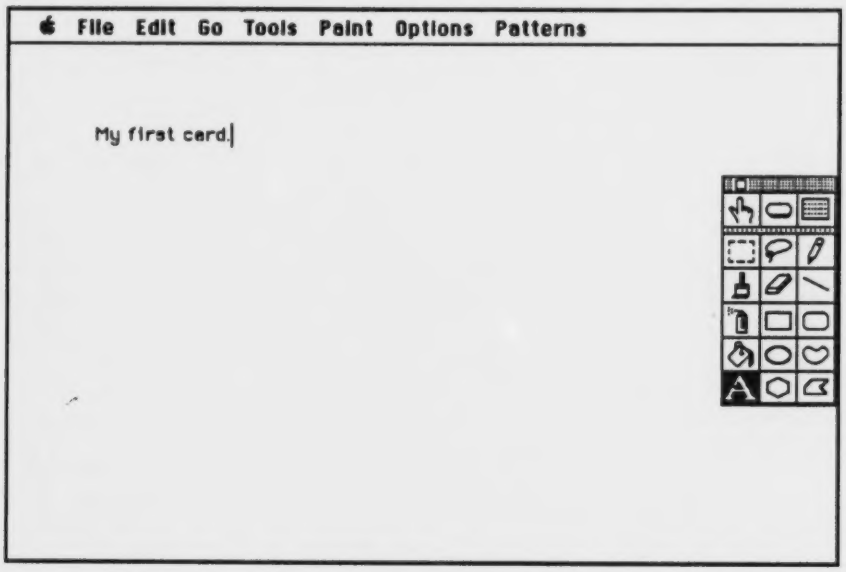
At this point you should give each of your cards a name. You should always choose something meaningful for the names. For now, call your cards "Demo 1" and "Demo 2." Recall that you name your cards by choosing **Card Info** from the **Objects** menu and then typing the name in the dialog box. You can select **Prev** or **First** from the **Go** menu to get to the first card and then select **Next** or **Last** to get back to the second card. You should get in the habit of ***always*** naming your cards as you create them. Using names for all of your cards will make your future work with HyperTalk much easier.

In order to tell the difference between these two cards, you can draw something on each one. Go to the first card. Be sure you have named your card. Next, choose **Tools**

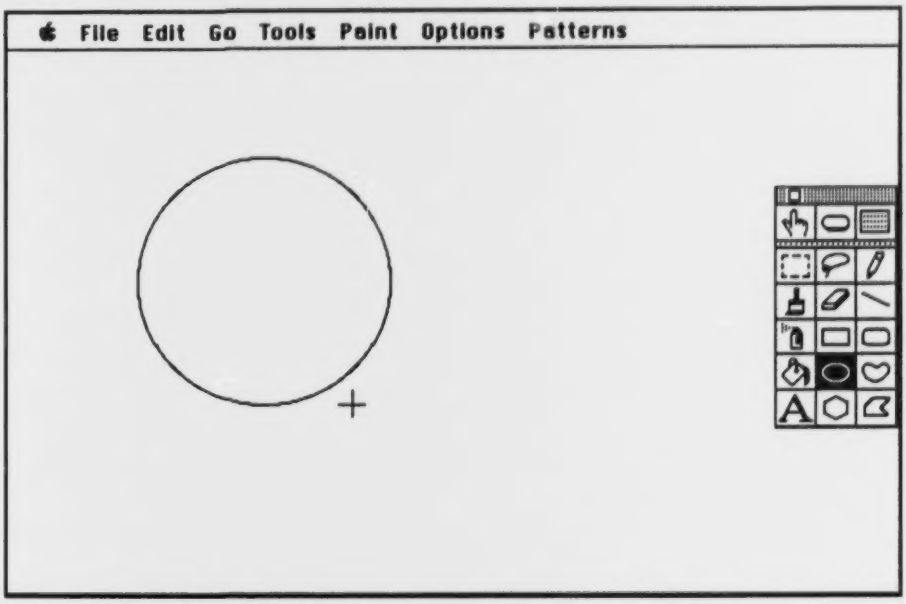
from the Menu bar and then choose the Text tool from the Tool window. (The Text tool is the letter "A" in the Tool window.)



You may prefer to "tear off" the Tool window so that you will always have it ready for use. Select **Tools** from the Menu bar, hold down the mouse button as you drag the window down to the location where you want the Tool window to be placed. You'll see a flashing rectangular outline move across the card as you move the mouse. When you release the mouse button, the Tool window will appear at the location of the mouse. Then select the Text tool from the newly placed Tool window. Put the words "My first card" on the left side of the card Demo 1. Note that the "text" that you put on the card with the Text tool is actually "painted" onto the card. That is, the letters behave like drawings, not like word-processed text.



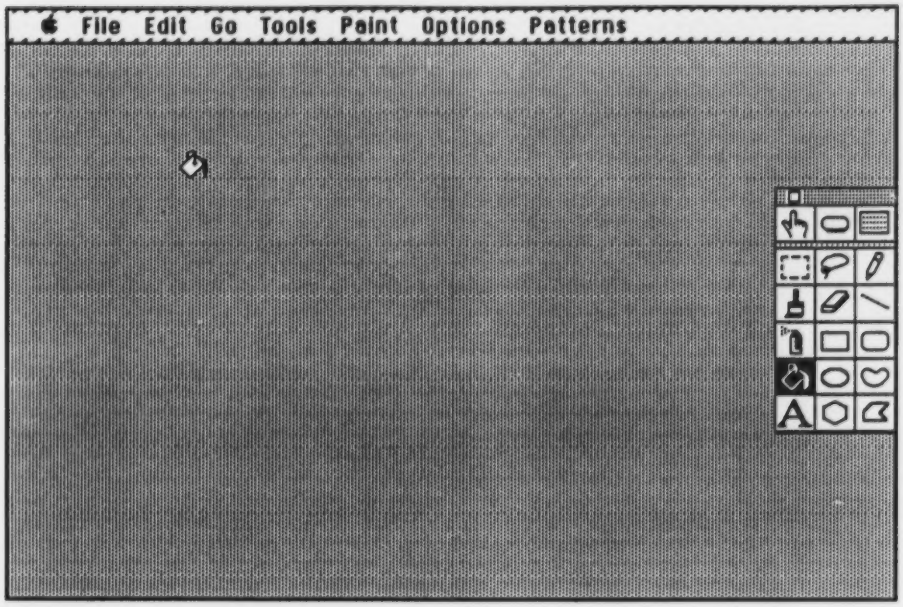
Select the Browse tool (the pointing hand) from the Tool window. Use the **Go** menu to go to your second card. Next, choose the Oval tool and draw a circle on the left side of the card:



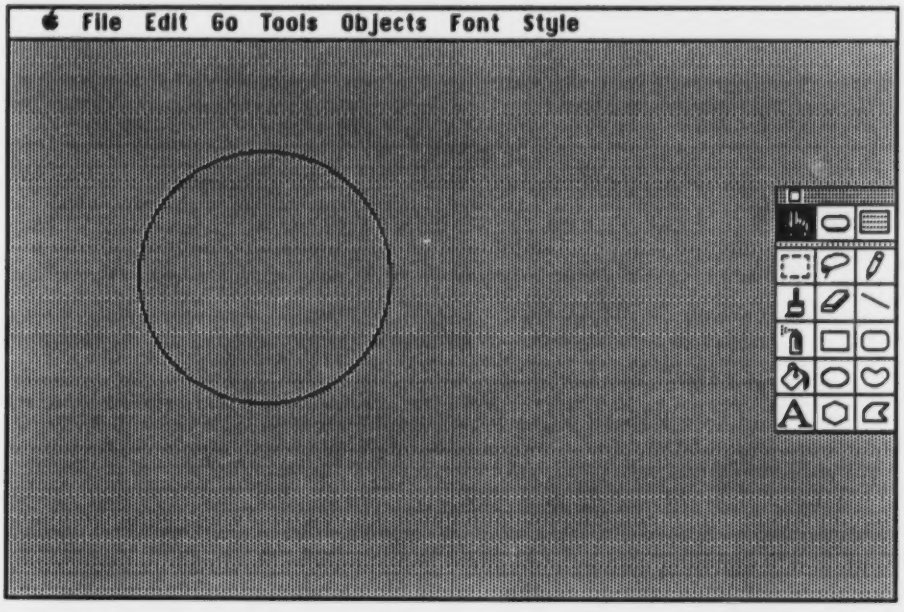
Now you can easily tell which card you are on as you move through your new stack. Note that if you continue to select **Next** from the **Go** menu, you will cycle through your two cards repeatedly.

A Brief Review—Editing a Background

You now have two cards in your stack, each with the same background. Select the **Background Info** under the **Objects** menu. You'll see that the background is shared by two cards. Now you are ready to modify the background. Select **Background** from the **Edit** menu. You see small diagonal lines along the Menu bar at the top of the screen. You do not see the circle or text that is on your cards. Select the Bucket tool and choose a pattern from the **Patterns** menu. Click anywhere on the background and the area fills with the pattern of your choice:



Next, select **Background** from the **Edit** menu again. You see your card, showing either a circle or some text "over" the new background:

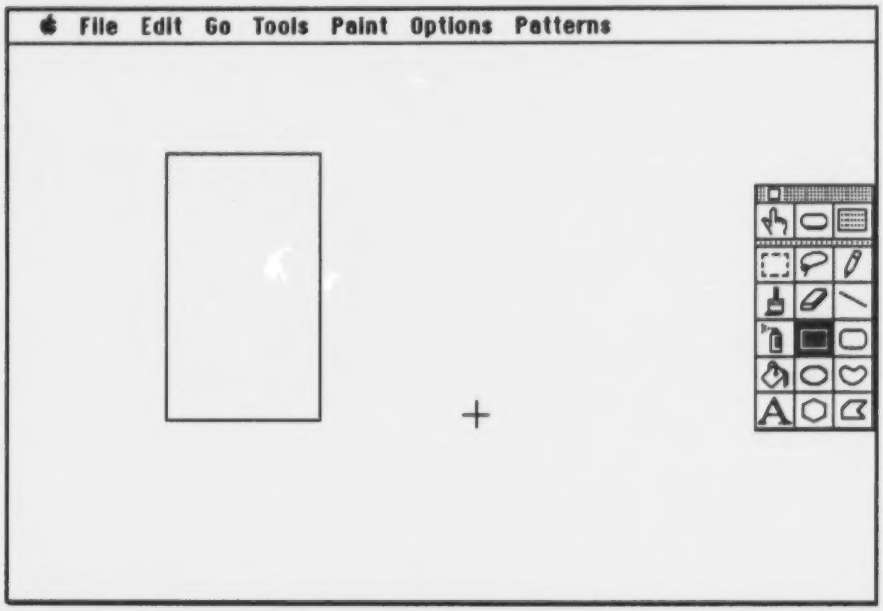


Check the other card in your stack. You see that both cards now have the same background.

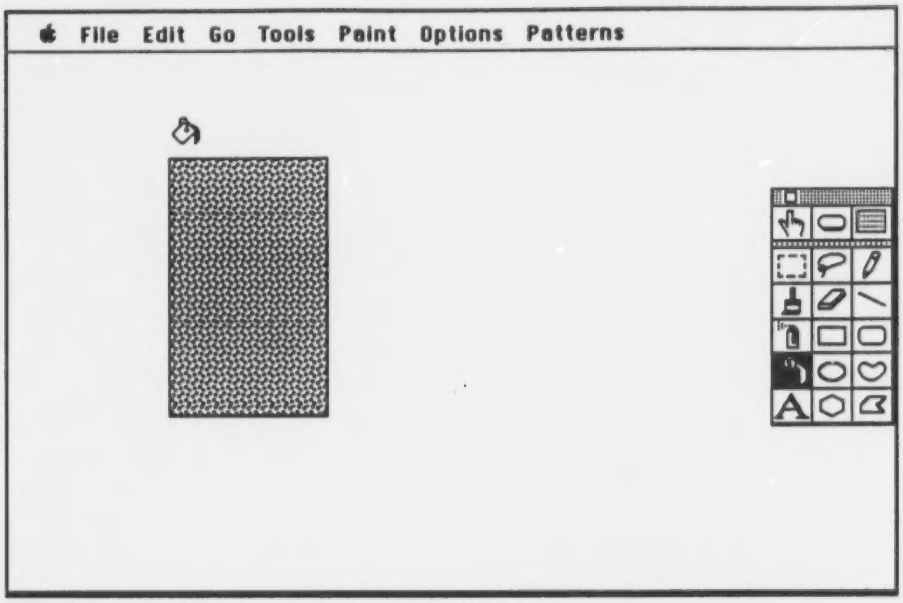
BEST COPY AVAILABLE

A Brief Review—Creating a New Background

You now have a stack containing two cards: Demo 1 and Demo 2. Demo 1 contains the words "My first card" while Demo 2 has a circle on it. Both cards have a shaded background. You can create a new background by selecting **New Background** from the **Objects** menu. You have just created a plain background *and an associated card*. Select **Background** again to move from the new background you have created to the associated new card. Name it "Demo 3." Use the Rectangle tool to put a rectangle on the left side of the card:



Next make a copy of this new card by choosing **Copy Card** from the **Edit** menu. Then choose **Paste Card**. Your screen doesn't appear to change, but if you check carefully, you'll see that your stack now contains four cards. Notice that when you copy a card, you get the current background as well as the exact contents of the current card, including the name of the card. Call your new card something like "Demo 4." Fill the rectangle on card Demo 4 with a pattern of your choice:



Now take a careful look at your stack. You can use key shortcuts instead of the **Go** menu. For example, pressing Command-1 moves you to the first card in your stack. You should see a shaded background with some text painted on the card. Next use Command-3 to go to the next card. You should see a shaded background with a circle on the card. Next comes a blank background with a rectangle, and finally, the last card in your stack has a blank background with a filled rectangle on the card.

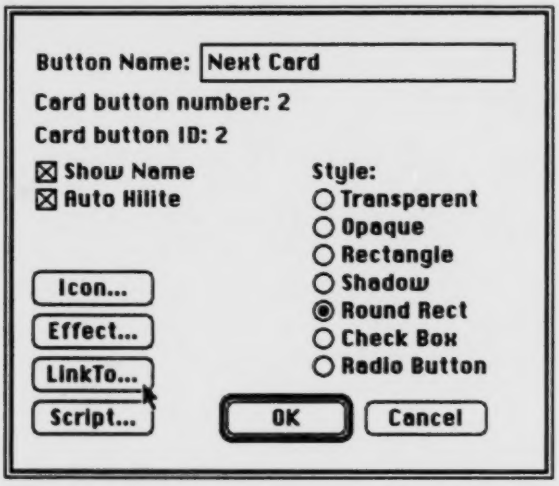
Does your stack have exactly four cards? Does each one have a name? Examine the **Card Info** dialog box (under the **Objects** menu) for each card. Be sure you understand exactly why the card looks the way it does. Examine the **Bkgnd Info** dialog box for each background.

Creating new cards and backgrounds can be confusing. When you create a new card, you get a new, blank card that has the same background as the card you were on *when you created the new card*. When you create a new background, you get both a new, blank background and a new, blank card. This can be confusing at first because blank cards and backgrounds all look the same on the computer screen.

A Brief Review—Creating a Button

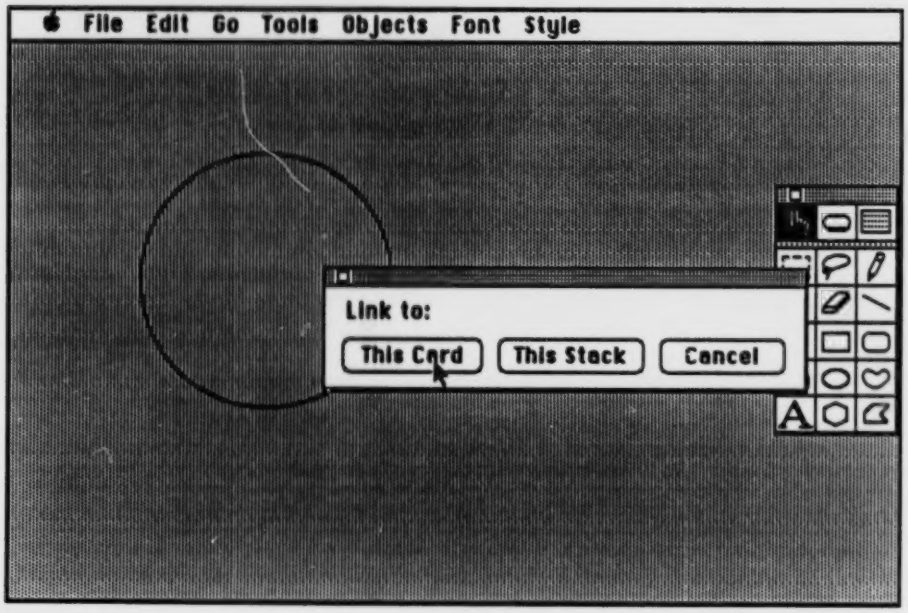
Buttons are commonly used to link cards so that you can easily move from one card to another without using the **Go** menu. You might want to create a button to move to the next card in the stack. Go to your first card. Choose **New Button** from the **Objects** menu

and then choose **Button Info** (or you can double click on the new button that appears). Next choose the form of your button:



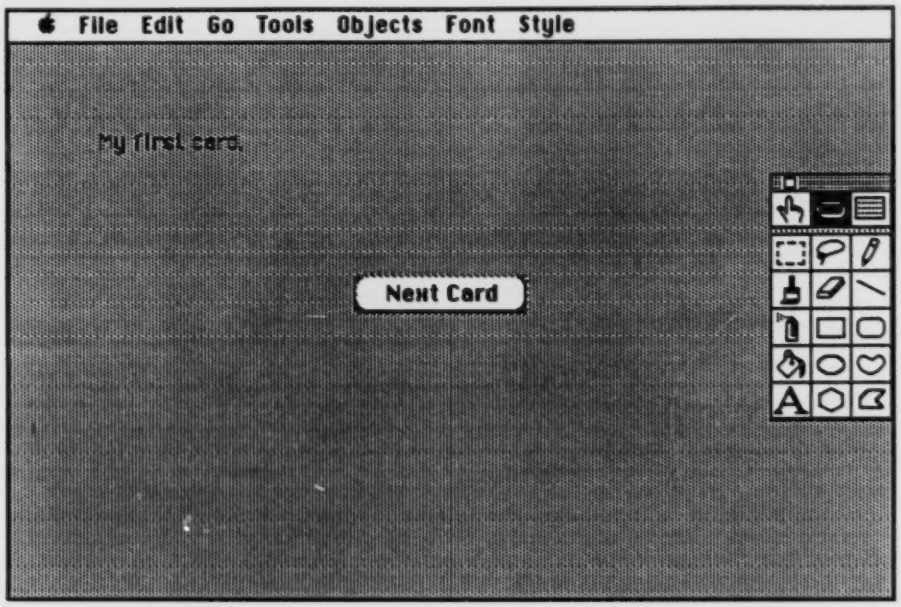
Note that if you are using a version of *HyperCard* before 2.0, the dialog box will look slightly different.

Be sure to name your button, perhaps "Next Card." You should always name your buttons in a meaningful manner. Finally, click on **LinkTo** and use the **Go** menu (or the appropriate Command keys) to go to the second card and establish a link:

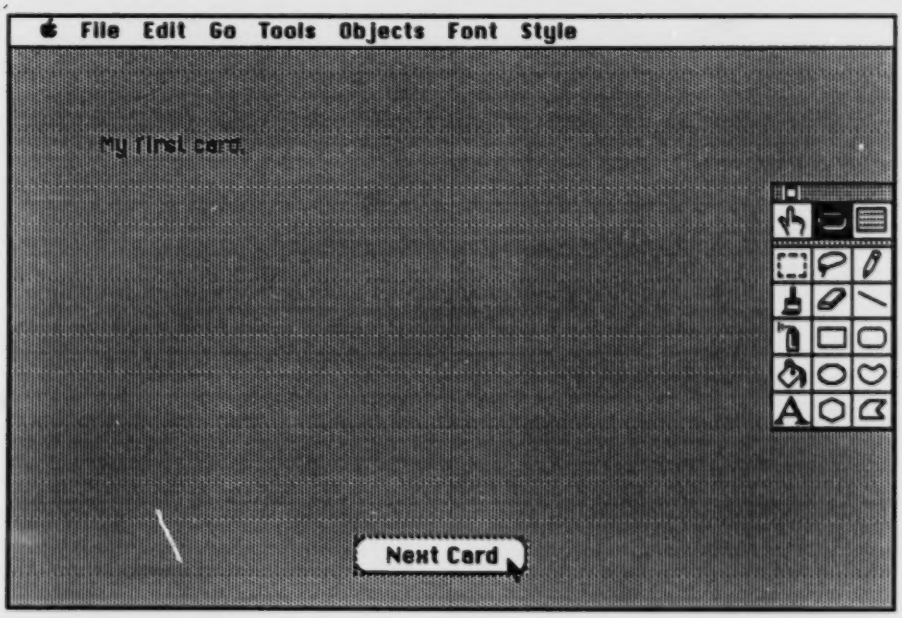


BEST COPY AVAILABLE

Your screen should now look like this:



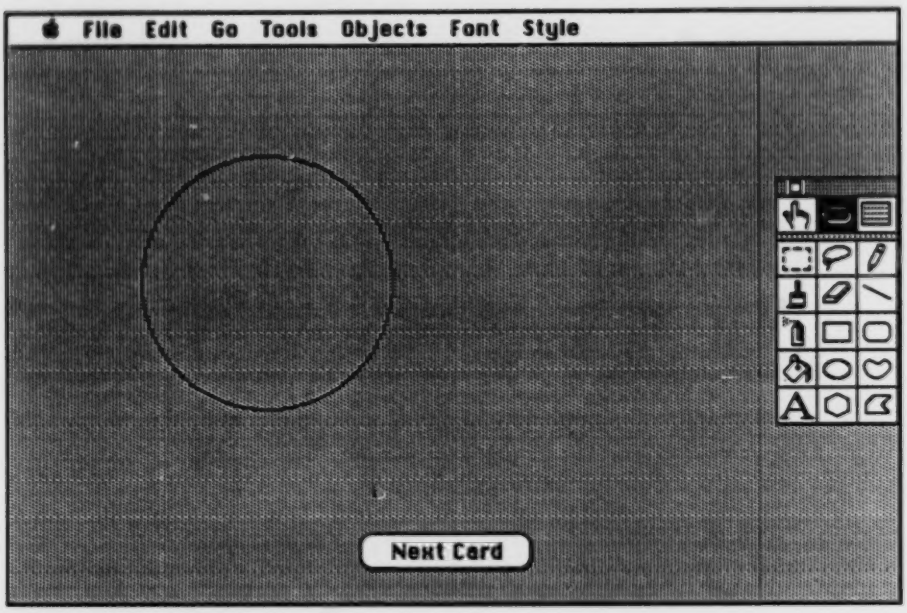
Click on the button and drag it to a convenient spot on the card, perhaps at the bottom center. Check to be sure that your new button works correctly. (Don't forget to choose the Browse tool before you try to navigate by clicking on the button.)



Perhaps you would like to have this button on every card. Be sure you are on card Demo 1. Simply select the Button tool (in the middle of the top row of the Tool window) and click on the button **Next Card**. You see a "box" appear around the Next Card button.

BEST COPY AVAILABLE

Select **Copy Button** from the **Edit** menu. Go to the second card. Select **Paste Button** from the **Edit** menu:



Note that the button appears on the second card in exactly the same position as on the first card. Finally, select **Button Info** from the **Objects** menu. Now practice copying the button and linking other cards in your stack.

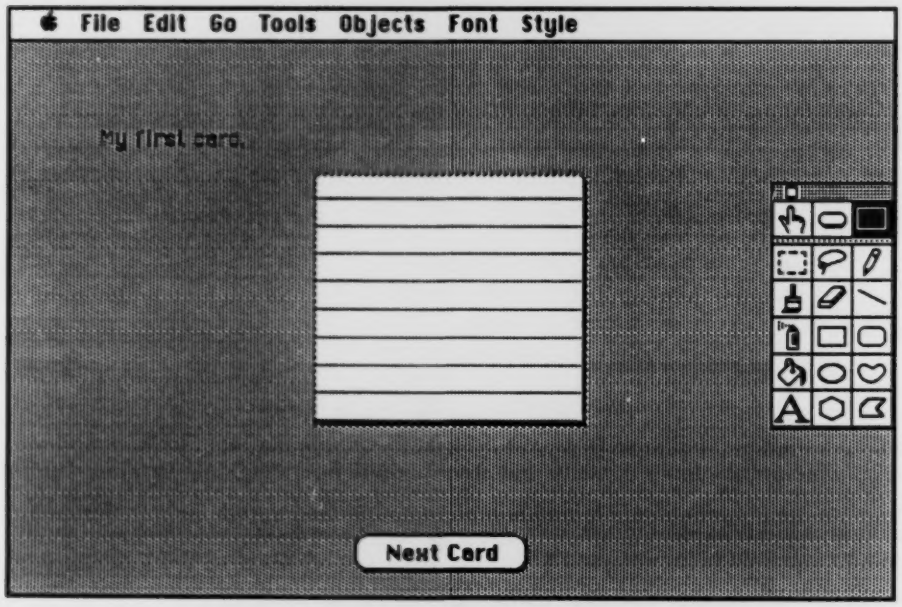
A Brief Review—Creating a Field

To create a field, you follow steps similar to those you used when you created a button. Again begin on card Demo 1. First choose **New Field** from the **Objects** menu. Then choose **Field Info** and decide on the design of your field:

BEST COPY AVAILABLE

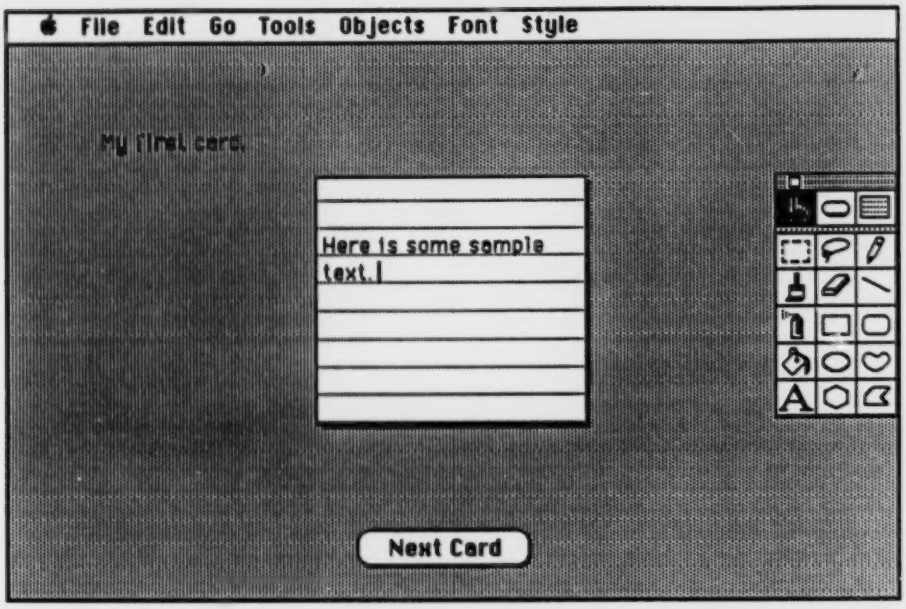
Technical Note: If you are using a version of HyperCard *before 2.0, the dialog box will look slightly different. Also note that if you use the background that comes with Version 1.2.5, there will be a narrow background field near the top of the card.*

Next, make your field any size by clicking near the edge of the field and dragging. Then you can place your field wherever you want by clicking near the center of the field and dragging it to the location of your choice:



Finally, use the Browse tool to place text in the field. Note that the "hand" changes to an "I-beam" when it enters the field. As you begin to type, the cursor then changes to a vertical bar:

BEST COPY AVAILABLE



Recall that the text you created earlier using the Text tool is "paint text." Such text is part of the graphics on the card. It can be changed only by using the tools below the top row of the Tool window. The text you entered in the field is text that can be edited like the text created using a word processor. Try placing the cursor or pointer in the middle of the line you just typed and adding some words. You see that you can easily modify the text in a field. Paint text, however, can only be modified by using the Paint tools such as the Select tool and the Eraser tool.

You now have a stack that contains four cards, several buttons, and a field. Your stack has been automatically saved as you were working. You do not need to save as you make changes in *HyperCard*. When you quit, *HyperCard* will store any changes you have made, so they will always be available next time you use the stack.

If you are not yet comfortable with the material in this chapter, you should spend some more time experimenting with creating your own *HyperCard* stacks. If you are comfortable with the material reviewed so far, you are ready to move on to the next chapter, where you will begin learning about scripting using HyperTalk.

BEST COPY AVAILABLE

Keyboard Shortcuts

Many of the things that you can do from the *HyperCard* menus can be done by using various key combinations. Below is a list of keystrokes that will come in handy as you work with *HyperCard*.

Navigation Between Cards	
~ (and `) key	Go back
Command-1	Go to the first card of current stack
Command-2	Go to previous card
Command-3	Go to next card
Command-4	Go to the last card of current stack
Command-H	Go to the Home stack
Command-R	Show the most recent cards

Editing	
Command-X	Cut
Command-C	Copy
Command-V	Paste
Command-Z	Undo
Command-N	New card
Command-B	Toggles edit background

Files	
Command-O	Open a stack
Command-Q	Quit *HyperCard*

Show/Hide	
Command-space bar	Hide/show menu bar
Command-Tab	Choose Browse tool
Command-Tab-Tab	Choose Button tool
Command-Tab-Tab-Tab	Choose Field tool
Command-Option	Display buttons (Click on a button to display the script)
Shift-Command-Option	Display fields (Click on a field to display the script)
Command-Option-C	Edit script of current card
Command-Option-B	Edit script of current background
Command-Option-S	Edit script of current stack

Chapter 1. Using Visual Effects

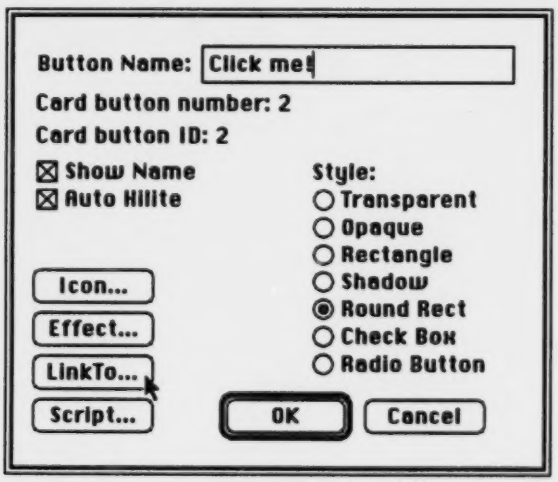
In this chapter, you will learn to write HyperTalk programs to *script* buttons. You will also learn to use special *visual effects* that are used to make a transition from one card to the next.

Introduction to Scripting

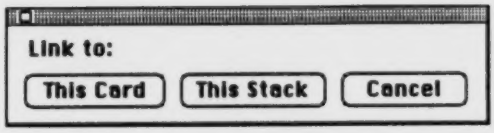
HyperCard makes it easy to create impressive effects on the Macintosh screen. In addition to putting both graphics and text on the screen and linking cards one to another, you can cause a number of different kinds of effects to appear as you move from card to card. Even if you have only created *HyperCard* stacks by using the pull-down menus, it is likely that you have experimented with visual effects. Using anything beyond the simplest visual effects requires that you learn about *scripting*. Scripting is the word used to describe writing a program in HyperTalk.

To get started exploring how to write simple scripts, create a new stack containing two cards. Name these cards "Demo 1" and "Demo 2." Be sure you are on the first card, Demo 1.

Next, create a new button on the card and give it a name, perhaps "Click me!" It is also a good idea to check **Auto Hilite** as well. With **Auto Hilite** turned on, people who use your stack will be able to see that they actually clicked on your button because the button turns black while the mouse button is down. Making use of **Auto Hilite** is a simple way to improve the "user friendliness" of your stacks. Click on the **LinkTo** button:



You see this dialog box:



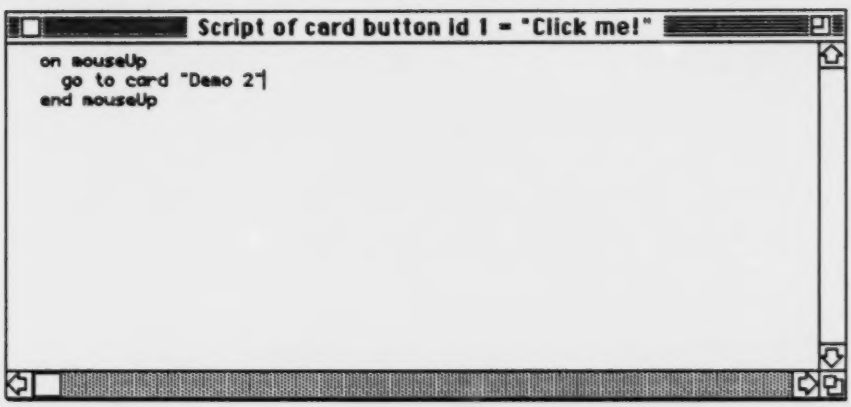
Use the **Go** menu or Command-3 to move to the card Demo 2. Then click on **This Card** to link the two cards. Now if you click on the Click me! button, you find yourself on the card Demo 2. Go back to the first card, select the button, and then select **Button Info** from the **Objects** menu. Click on the **Script** button.

You see a screen that contains a script similar to this one:

```
on mouseUp
   go to card id 6176
end mouseUp
```

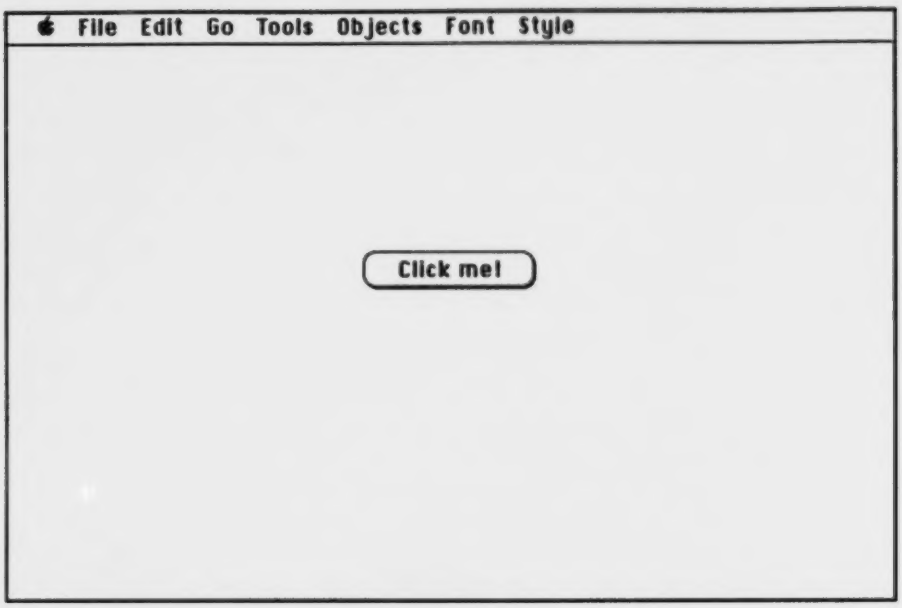
HyperCard automatically created this script for you when you clicked **LinkTo**. This script includes the **go** command, which you will learn more about later in this chapter. Generally, it is a good idea to write your scripts using the names of your cards. Change the script so that the **go** command-line reads

```
on mouseUp
   go to card "Demo 2"
end mouseUp
```



Notice that there are quotation marks around the name of the card. You should always put quotation marks around the name of an object in HyperTalk. It helps avoid confusion.

Next, check that your script works by clicking on your Click me! button:



Note that there are several ways to get to the script of a button. You can select the Button tool, and then choose **Button Info** under the **Objects** menu and click on the **Script** button. Or you can select the Button tool, and double click on the button and then click on the **Script** button in the dialog box. Or you can simultaneously hold down the Command and the Option key and click on the button. This last method works when either the Browse tool or the Button tool is selected, and it takes you directly to the script.

In the previous discussion, the word "script" was used to describe programs written in HyperTalk. HyperTalk scripts are actually collections of *handlers*. The previous script,

```
on mouseUp
   go to card "Demo 2"
end mouseUp
```

contains one handler. The first line of a handler always begins with the keyword **on** (or the keyword **function**). The last line of a handler always begins with the keyword **end**. The lines in between, in this case just the line

```
go to card "Demo 2"
```

are called HyperTalk *statements*. Each statement ends with a Return character that is not visible.

When Macintosh is running, it is constantly checking for events such as the click of a mouse button or a key press. When the Macintosh notes an event, it notifies *HyperCard*. That is, the Macintosh tells *HyperCard* whether the mouse button is up or down, where the mouse is located, which key has been pressed, and so forth. *HyperCard* then sends "signals," or messages, about these events to various objects in a stack making use of the HyperTalk programming language. When you write handlers, you are telling *HyperCard* how to use these messages. In the previous script, **mouseUp** is a message handler built into HyperTalk. You will learn more about message handlers in Chapter 11 and more about message passing in Chapters 7 and 17.

"Handler" is actually short for "message handler." Handlers take care of—handle—messages sent when you are using *HyperCard*. "The mouseUp" handler takes

care of a "mouseUp" message that is sent to the card when you release the mouse button. When you start a new script for a button, a "mouseUp" handler containing no statements is automatically provided. However, you can use other handlers. For example, you might want to try "mouseWithin." "MouseWithin" responds whenever the mouse pointer is over the object containing the handler, in this case the button Click me! Change the script you have written so that it reads

```
on mouseWithin
   go to card "Demo 2"
end mouseWithin
```

Now whenever you move the mouse pointer over the button, *HyperCard* immediately moves to card Demo 2.

Have you noticed the unusual use of lowercase and uppercase letters in HyperTalk, especially in the handler names? To make the HyperTalk code more readable, the convention is to capitalize the first letter of the second and following words. Thus you usually see "mouseUp" rather than "mouseup." HyperTalk code will work whether you use uppercase or lowercase letters. However, this book will generally follows this conventional use of lowercase and uppercase letters.

Working with this new handler can be frustrating because it moves you away from card Demo 1 each time you move over the button Click me! You may find yourself constantly having to move back to your first card to experiment some more. A good command to use for such experimentation is the **beep** command. Thus, you might modify the previous script by entering

```
on mouseWithin
   beep 3
end mouseWithin
```

Now each time the mouse pointer touches the button, three beeps occur.

Technical Note: If your computer has some sound other than the conventional "beep" sound, that sound will occur three times. Note also that if the sound is turned off on your Macintosh, the Menu bar will flash three times.

As indicated earlier in this book, HyperTalk is a programming language that is part of the *HyperCard* application. Programming languages have a structure and set of rules that define the correct way to use them, just as English grammar rules define how to use English. As you progress through this book, you will learn the general rules—or *syntax* —for using each new command or function. In Chapter 7 you will learn more about the general structure of the HyperTalk language. It is important that you progress beyond memorizing and using specific examples of HyperTalk scripts. You should make an effort to understand the structure of each new command or function as it is introduced.

In this chapter you have seen the **go** command and the **beep** command. The correct way to use the **beep** command—its syntax—is given in the following box. (Note that the syntax of commands will be shown in plain rectangular boxes like the following one.)

```
beep [number]
```

The word *number* represents an unsigned integer (technically, an unsigned integer expression). Notice that the word "number" is in square brackets. Words in square brackets in the formal description of a HyperTalk command or function indicate that those parts of the command are optional. That is, you can type

```
beep
```

and hear one beep-sound, or you can type

```
beep 5
```

to hear five beep-sounds. "Number" is a *parameter*; it represents a value used by **beep**. Since "number" is not always needed, "number" is an *optional parameter*.

> *Technical Note: In some sources, **beep** is called a* HyperCard *"language element" and number is called a "general element." These sources say that you must substitute "actual instances" for general elements when using the command. The concept of substituting a value (an actual instance) into a general element is exactly the same as substituting a value for a parameter. In this book we will use the term "parameter," which is more commonly used in computer science settings.*

Similarly, the **go** command in its simplest form has the syntax

```
go [to] cardDescriptor
```

There are more complex forms of the **go** command, but they will be explained in Chapter 7. The word "to" is optional because it is in square brackets. "To" is an optional part of the command; it is *not* a parameter.

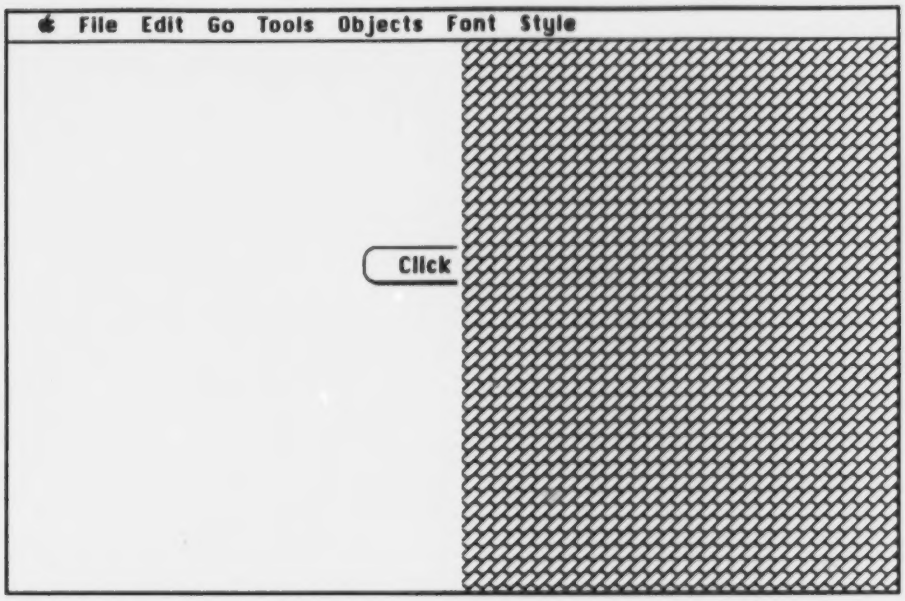
When you see italics in a syntax description, you will substitute a value for the italicized word when you use the command. Since parameters always have values substituted for them, they are always shown in italics. In other instances an italicized word in a syntax description represents a number, word, or group of words. For example, in the above description, the term *cardDescriptor* refers to the word "card" followed by a group of letters and/or numbers that represents a legal identification of a particular card.

Visual Effects

If you have looked at *HyperCard* stacks produced by other people, you have no doubt seen a variety of visual effects. If you have done much work with *HyperCard*, it is likely that you have experimented with visual effects yourself. Visual effects are flashy and easy to use. For example, modify the previous script on the Click me! button as follows:

```
on mouseUp
   visual effect scroll left
   go to card "Demo 2"
end mouseUp
```

Be sure you don't remove the "go to" statement. Visual effects only occur when you move from one card to another. Clicking on the button makes the first card appear to slide left so that you can see the second card. The effect is even more dramatic if the second card is quite different from the first. Put some graphics on the second card. You might even use the Bucket tool to fill the card with a pattern. Then the visual effect might look something like this:



Technical Note: If you are using a Macintosh with a color monitor and HyperCard *version 1.2.5, you must turn off the color in order for visual effects to appear. Select the* **Control Panel** *from the* **Apple** *menu. Click on the Monitors icon. Select Black & White instead of the Color mode.*

Another visual effect you might like to try is a "dissolve":

```
on mouseUp
   visual effect dissolve slowly
   go to card "Card 2"
end mouseUp
```

Now the first card "fades" into the second when you click on the button. Quite a variety of effects are available for you to use when working with *HyperCard*. However, you should resist the temptation to use these effects just because they are "neat." A discussion of appropriate uses of visual effects is given in Chapter 9.

After some work with visual effects, most people become bewildered by the many choices and have trouble deciding what is "legal." If you understand the structure of the **visual** command, it is easy see what legal combinations you can use to create interesting new visual effects. Here is the syntax of **visual**:

```
visual [effect] effectName [speed] [to image]
```

Now you can see that the *command* is really **visual**. The word "effect" is simply optional. That is, you can write

```
on mouseUp
   visual dissolve slowly
   go to card "Card 2"
end mouseUp
```

and the same effect will occur.

Technical Note: In some reference sources, effect is called an "optional element" or "helper word."

Next comes a required parameter, *effectName*. *EffectName* can be any one of the following:

```
barn door close        barn door open        checkerboard
iris open              iris close            venetian blinds
zoom close             zoom open             dissolve
zoom in                zoom out              plain
scroll down            scroll up
scroll left            scroll right
wipe down              wipe up
wipe left              wipe right
```

If you are using *HyperCard* 2.0, you can also use the following *effectNames:*

```
shrink to top                  shrink to bottom
shrink to center               stretch from top
stretch from bottom            stretch from center
cut
```

In the syntax for **visual**, there are two optional parameters. *Speed* is the first optional parameter and can be one of the following:

```
fast
very fast
slow[ly]
very slow[ly]
```

In the previous Click me! script, "slowly" was used. Try changing "slowly" to "very slowly."

The next phrase–[to *image*]–is also optional. While the entire *phrase* is optional, the "to" *must* be used if you use a value for "image." *Image* can be one of these terms:

```
black
gray (or grey)
inverse
white
card
```

Experiment with such scripts as

```
on mouseUp
   visual effect dissolve slowly to black
   go to card "Card 2"
end mouseUp
```

You can create some very interesting effects using "image."

Debugging and Testing

As you begin your work with *HyperCard*, it is easy to get lost or confused. Keep in mind that you can always use the **Objects** menu to give you information about your location. Simply select **Card Info** from the **Objects** menu and you will be able to identify the card on which you are working. In addition, you can use the **Go** menu or the keyboard shortcuts to get to the beginning (Command-1) or end (Command-4) of your stack, or you can return to the Home Card (Command-H) to start over.

Another useful navigational tool is **Recent** (Command-R) found in the **Go** menu. **Recent** shows you small pictures of the cards you have most recently visited. You can click on any of these pictures in order to go to that card. This option will even let you move among stacks. If, for example, you have retrieved a picture from another stack and pasted it in your new stack, you can easily return to the same spot in the stack containing the picture to retrieve something else. Simply click on the appropriate card picture after selecting **Recent**.

Visual effects can occasionally cause some confusion as well. Visual effects can only occur when you move to another card. So, if you have several visual effects in a button script, they will all happen only when you actually move to another card. If your button script doesn't include a **go** command, you will not see your visual effects when you click on the button, but you may see them later when some other action takes you to another card.

Here is another problem you might encounter with visual effects. Suppose you create the handler

```
on mouseUp
   visual effect zoom out
   beep 5
   visual effect zoom in
   go to card "Demo 2"
end mouseUp
```

Your intention is that the user of your stack will see a "zoom out" effect, hear five beeps, and then see a "zoom in" effect. In actual fact, the user will first hear five beeps and then see the two visual effects, one after the other. Keep in mind that visual effects occur only on card transitions. Also recall that if you are using *HyperCard* 1.2.5, you must turn off the color on a color monitor in order to see visual effects.

Note that visual effects occur at different speeds, depending on the model of Macintosh you are using. That is,

```
visual effect dissolve slowly
```

will be much slower on a Macintosh Plus than on a Macintosh SE/30.

New HyperTalk Words in This Chapter

Objects
background
button
card
field
stack

Object Types
backgrounds
buttons
cards
fields

Commands
beep
go [to]
visual

System Messages
mouseUp
mouseWithin

EffectNames (used with visual)	
barn door close	barn door open
iris open	iris close
zoom close	zoom open
zoom in	zoom out
scroll down	scroll up
scroll left	scroll right
wipe down	wipe up
wipe left	wipe right
checkerboard	venetian blinds
dissolve	plain
If you are using *HyperCard* 2.0, you can also use:	
shrink to top	shrink to bottom
shrink to center	stretch from top
stretch from bottom	stretch from center
cut	

Speed (used with visual)
`fast`
`very fast`
`slow[ly]`
`very slow[ly]`

Image (used with visual)
`black`
`gray`
`white`
`inverse`
`card`

Project Ideas

There is an almost endless number of ways to combine the various parts of the **visual** command. Spend some time experimenting. Do you see what each effect does? How does the value of *speed* change each effect? What effects are caused by different values for *image*? Begin your exploration with two dramatically different cards, like the previously described Demo 1 linked to Demo 2, which is filled with a pattern. Then try the same experiment with two quite similar cards. How does the "feel" of each effect change?

Spend some time experimenting with scripting buttons to produce dramatic visual effects combined with beeps. You might try some effects such as

```
on mouseUp
   beep 2
   visual effect iris open to black
   visual effect iris close
   go to card "Demo 2"
end mouseUp
```

As you are experimenting, think about the "feeling" provided by each effect. What kinds of visual effect combinations would you use to present various kinds of information?

Create a stack to demonstrate all of the visual effects, including perhaps some more complex or flashy effects. You should be able to see what each effect looks like when moving between both similar and different cards. Such a stack could be an excellent group project, with each group creating a series of buttons, all of which would then be copied onto one card designed by yet another group.

Take some time to think about what you have learned about HyperTalk syntax. Here is a command you have not seen before:

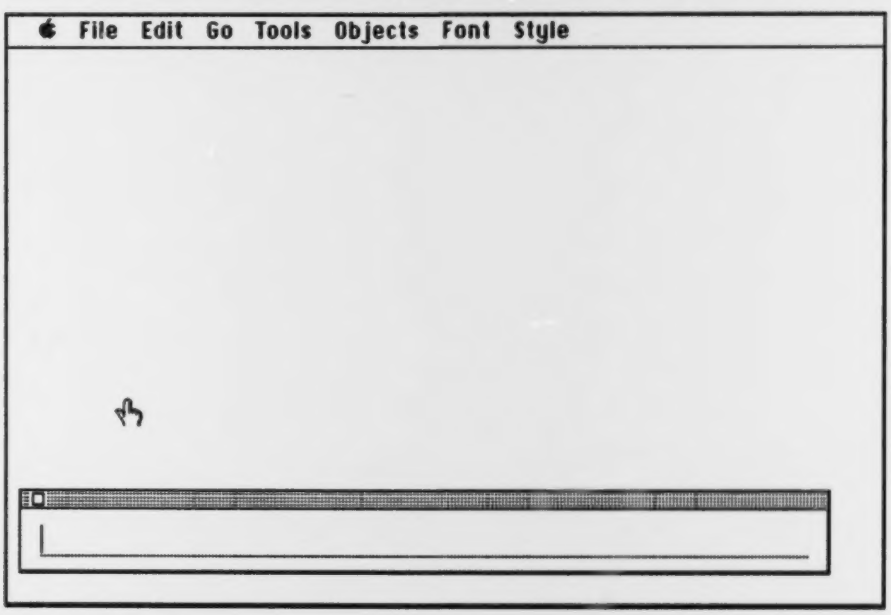
```
wait [for] integer [seconds]
```

Can you identify the command? What is the word "for" called? Which word(s) are parameters, if any? What is "seconds"? Experiment with **wait**. How might you use it when writing a handler for a button? (Did you identify **wait** as a command, "for" and "seconds" as optional elements, and "*integer*" as a parameter?)

Chapter 2. Boxes, Windows, and More

In this chapter, you will learn about the Message box and how to use it to run HyperTalk statements. In addition, you will learn how to manipulate the Card, Tool, and Pattern windows using HyperTalk. Finally, you will learn how to use the functions **date, time, seconds,** and **ticks.**

In the last chapter you learned how to write HyperTalk scripts. These scripts, made up of one or more handlers, were all on buttons. They caused some action to occur when you clicked on the button. You used these HyperTalk commands only when they were part of a script. It is also possible to use HyperTalk commands without writing a script. This is done by using the Message box. If you pull down the **Go** menu, near the bottom of the menu you see **Message.** If you select **Message** or press Command-M, you see



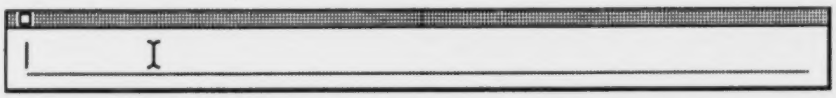
The Message box appears at the bottom of your current card with a vertical bar pointer (|) at the left side.

Technical Note: If you are using a screen larger than the "standard" 9-inch Macintosh screen, the Message box may appear below your card. Click in the gray area at the top of the Message box and drag it onto the card as shown in the previous illustration.

Place the Browse tool as shown in the previous figure. Slowly drag the mouse straight towards the bottom of the screen. You see the pointer change from the pointing hand to the arrow to the I-beam, back to the arrow, and finally back to the pointing hand. Whenever you are over the center of the Message box, you will see the I-beam appear. You do not have to be using the Browse tool; no matter what tool you have

BEST COPY AVAILABLE

selected, the I-beam appears when you are over the area in the Message box where you can type.



The Message box is a special kind of one line *container*. You will learn more about containers in Chapter 23, but for now you only need to know that you can type HyperTalk commands in the Message box. For example, place the mouse pointer over the Message box and type

```
go to next card
```

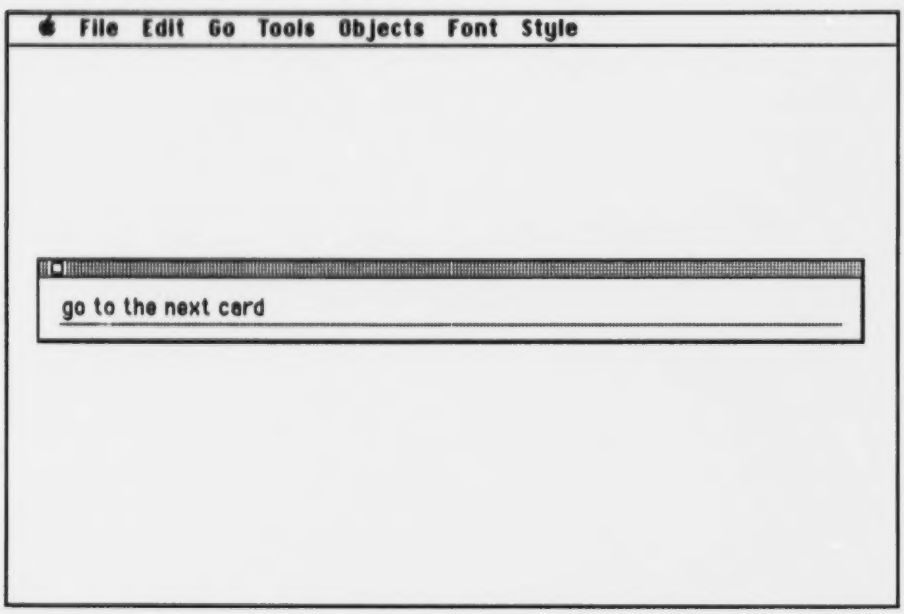
When you press Return, you are on the next card. The word "next" can always be used to refer to the next card in your stack. If you are using *HyperCard* 2.0, you can use

```
go to the next card
```

to make your code more like standard English. Note that if you have only one card in your stack, this command will have no effect.

Note that when you move to the next card, the Message box appears in the same place on your destination card. Use the **Go** menu, Command-3, or the Right Arrow key to move through your stack. The Message box is at the same place on every card in your stack.

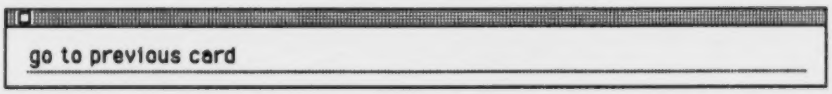
You can place the Message box anywhere on your screen. Move the mouse pointer over the shaded bar at the top of the Message box. Drag it to the location of your choice by holding down the mouse button and releasing it when the Message box is where you want it:



Note that it appears in the same position on all of your cards. Anything you type in the Message box on one card will still be there when you move to another card.

You can change the contents of the Message box by using the mouse and keyboard to edit what is already there. For example, you might replace "next" with "previous" to go to the previous card. You can double click on "next":

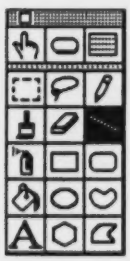
and simply type "previous" or "prev" in its place:



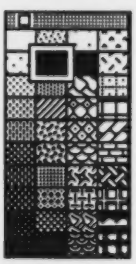
If the vertical bar is flashing in the Message box, any text you type will be inserted after the location of the pointer. If you want to completely replace the text in the Message box, you can select it all and press the Delete key. You can also move the mouse pointer outside the Message box and click anywhere. When you begin to type, the Browse tool disappears and the text you type appears in the Message box, leaving the flashing vertical bar to the right of the text.

Windows

When you are working with *HyperCard*, you use a number of *windows*. You have certainly used the Tool window:



Perhaps you have also used the Pattern window. After you select any Paint tool (below the dotted line in the Tool window), you can select **Pattern** from the Menu bar. This gives you a tear-off window that can be placed anywhere on your cards:



The Message box and the card itself are also windows. These windows all have *properties*, such as "size," "location," and "visible." You will learn more about properties in Chapters 4 and 8. For now, you'll have the opportunity to briefly explore window locations and determine whether a window is visible.

Technical Note: In some sources, the Tool window and the Patterns window are referred to as the Tools palette and the Patterns palette, respectively. Since you must use the word "window" to refer to these two objects in HyperTalk, this book will not use the word "palette." While the Message box is also a window, HyperTalk allows you to refer to it either with the word "window" or with the word "box." This book will use the term "box" since the Message box is significantly different from the other windows used in HyperCard.

Show and Hide

Make sure the Tool window and the Message box are on your screen. In the Message box, type

```
hide the tool window
```

The Tool window disappears. Next type

```
show the tool window
```

in the Message box. The Tool window reappears.

Show and **hide** are new HyperTalk commands. In its simplest form, their syntax is:

```
show [the] windowName [at integer1, integer2]
hide [the] windowName
```

Notice that **show** and **hide** have a number of options, much as **visual** does. In these descriptions, "windowName" is a parameter. You must provide values for those parameters. The phrase "[at *integer1,integer2*]" in the **show** command is optional. In the **show** and **hide** commands, *windowName* can be any one of the following:

```
card window
tool window
pattern window
[the] message (or msg) [window]
[the] message (or msg) [box]
```

Take a moment to look carefully at this list. Notice that there are many ways to refer to the Message box. For example, the following are all equivalent:

```
show the message window
show message window
show msg window
show the message box
show message box
show msg box
show msg
```

You can refer to the Message box in this wide variety of ways in most HyperTalk statements. Also note the optional phrase in **show**. Here "integer1" and "integer2" represent horizontal and vertical coordinates. (See the example in the following illustration.) Also note that although the last phrase of **show** is optional, "at" must be used when you want to include the two coordinates.

The Message box is an excellent tool to use when experimenting with new HyperTalk commands. You don't have to create objects and write entire scripts. You can simply type the commands in the Message box. For example, you might type

```
hide the pattern window
```

or

```
show the tool window
```

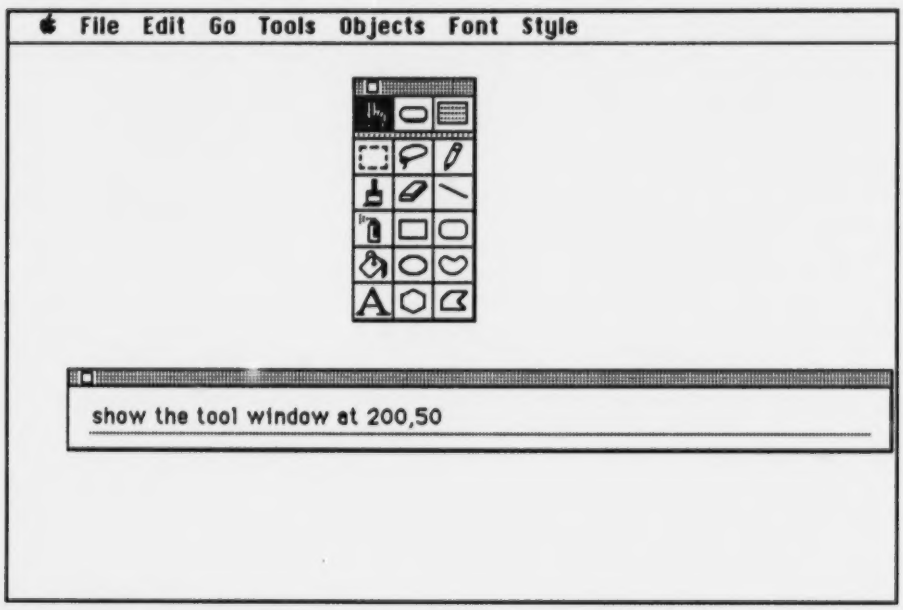
Note that "the" is optional with **show** and **hide**. You might experiment with changing the location of some of the windows. For example, if you type

```
show the message box at 100, 200
```

in the Message box, followed by

```
show the tool window at 200, 50
```

you see



Technical Note: The coordinates used after **show** *indicate the position of the upper left corner of the Tool window or Message box* below the shaded bar *along the tops of the windows.*

You can also type

```
hide the message box
```

to cause the Message box to disappear. Typing "hide the message box" in the Message box has exactly the same effect as clicking in the Close box in the upper left corner of the Message box. Once you hide the Message box, you must use either Command-M or the **Go** menu to get it back.

Spend some time experimenting with these commands. Can you figure out the coordinates of your screen? How wide is the Message box? How tall is it? Where is the center? Where is the origin (the point 0,0)? To begin such an experiment, try typing

```
show the message box at 0, 100
```

Then, gradually move the Message box to the right:

```
show the message box at 50, 100
show the message box at 100, 100
```

How far to the right can you move it? Next, change the second coordinate. How far up and down can you move it?

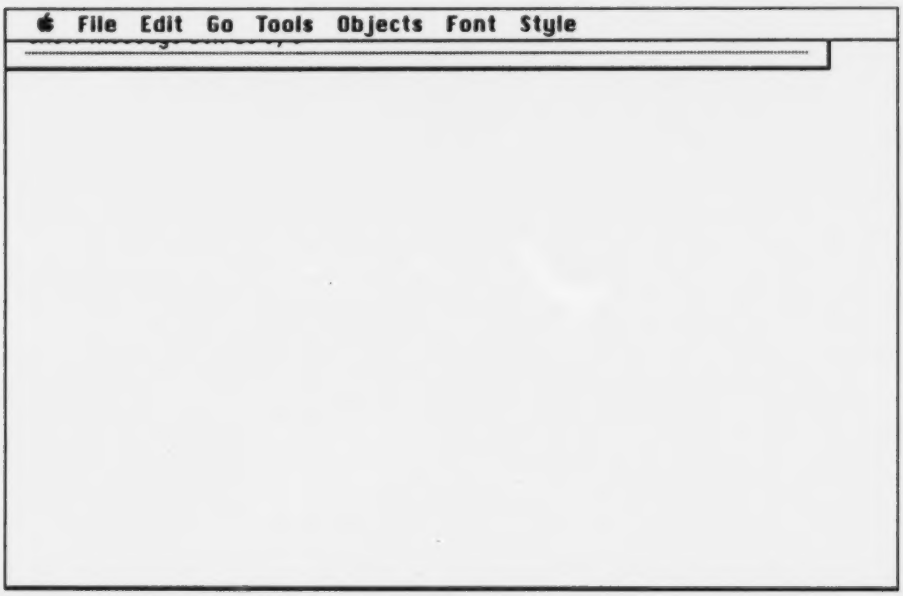
Technical Note: The dimensions of the "standard" Macintosh screen are 512 pixels across and 342 pixels down. This is the size of standard HyperCard *cards as well. However, in* HyperCard *2.0, cards may vary in size.*

More on Show and Hide

As you were trying to find the top-most coordinates of your screen, did the Message box disappear under the Menu bar? For example, if you type

```
show the message box at 0,0
```

you see



Perhaps you remembered that Command-Space bar makes the Menu bar disappear? Another solution to getting the Menu bar out of the way is to type

```
hide the menuBar
```

in the Message box. Here you see that **hide** and **show** can also be used with "menuBar":

```
show [the] menuBar
hide [the] menuBar
```

If you type

```
show the menuBar
```

the menu bar reappears. If you are looking at someone else's stack that has the Menu bar hidden, you can always use the Message box and the **show** command to make the Menu bar visible.

Hide and **show** can be used even more broadly. They can be used to make *HyperCard* objects appear and disappear. Suppose you have a card containing the button Click Me! If you type

```
hide card button "Click Me!"
```

in the Message box, the button disappears. You can use

```
show card button "Click Me!"
```

to make the button visible again.

Another form of **hide** and **show** is:

```
hide objectName
show objectName [at integer1,integer2]
```

Here, *objectName* is a parameter that can have any of the following values:

```
[card] button string
background button string
card field string
[background] field string
```

String represents the name, number, or ID of the button or field.

There is yet another quite useful version of **show**. It has two forms:

```
show integer cards
show all cards
```

Typing this command in the Message box lets you "flip through" all or part of your cards. Try it!

Technical Note: If you are on a "fast" Macintosh, the cards flip by very quickly. In Chapter 24, you will learn programming techniques that will allow you to flip through cards more slowly .

The Message Box as a Container

You have seen that you can type HyperTalk commands in the Message box. This can be useful when testing single-line HyperTalk statements. In addition to its use for typing single-line statements, the Message box can be used to hold information. HyperTalk uses the term *container* for objects that hold information. (You will learn more about containers in Chapter 23.) The Message box is a special container that can hold only one line at a time.

In addition to HyperTalk commands, there are also HyperTalk *functions*. HyperTalk commands cause some action to occur, such as moving to another card or creating a visual effect. Functions produce values. If you type the function

```
the date
```

in the Message box, the current date appears in the Message box. If you don't give HyperTalk any other directions, it automatically puts the result of a function into the Message box. Functions automatically make use of the Message box as a container to hold information.

The syntax of **date** is as shown in the following double box. Note that functions will be shown inside a double box; recall that commands are shown with a single box around them.

```
the [adjective] date
```

Here *"adjective"* must be one of the following:

```
long
short
abbreviated (or abbrev or abbr)
```

So, you might type:

```
the long date
the abbreviated date
the short date
```

Have you noted that you must use the word "the" with **date**? Try typing just

```
date
```

in the Message box. You will get an error message. **Date** must have the word "the" included, as indicated in the previous description of its syntax. Experiment to see what changes "long," "short," and "abbreviated" make in the date that appears in the Message box.

There are three other functions in HyperTalk that deal with time. They are **seconds, time,** and **ticks**. Here is the syntax for each:

```
the ticks
ticks()
the seconds
seconds ()
the [adjective] time
time ()
```

Seconds returns the number of seconds between January 1, 1904, and the current time set on your Macintosh. (This somewhat strange choice of date was established by Apple Computer, Inc. and cannot be changed.) **Ticks** returns the number of 60ths of a second since your Macintosh was turned on, and **time** returns the current time—assuming your Macintosh clock is set correctly. Note that *"adjective"* is optional, and, like **date,** can have the values "long," "short," and "abbreviated." However, **time, short time,** and **abbreviated time** all return the same form of the current time. Also note the second form of these three functions. You can omit the word "the" if you include parentheses after the name of the function. (There are a few functions with which you cannot use "the." You will learn about these in Chapter 22.)

Thus, if you type

```
the ticks
```

in the Message box, you will see something like

```
467573
```

a number that represents the number of 60ths of a second since you turned your machine on. If you type

```
time()
```

in the Message box, you might see

```
4:14 PM
```

while typing

```
the long time
```

might put

```
4:14:38 PM
```

in the Message box.

Functions in Scripts

Now try another experiment. Put the following handler on a button that is on your card:

```
on mouseUp
   the date
end mouseUp
```

Try it. What happens? Why did "the date" work in the Message box and not when you put it in a handler?

Remember that **date** is a *function*. A function returns a value that is then used by *HyperCard*. When you type a function in the Message box, the value returned is automatically displayed there. However, in the previous handler, HyperTalk doesn't know what to do with the value produced by "the date." You can modify the handler as follows:

```
on mouseUp
   put the date into the message box
end mouseUp
```

Now when you click on your button, the current date appears in the Message box.

Put is a new and very useful command. The syntax of **put** is:

```
put expression [preposition container]
```

An *expression* in HyperTalk describes how to get a value. An expression might be simple, such as a single number or name, or it might be quite complex. *Preposition* in this case must be "into," "before," or "after." *Container* is the name of some container, such as the Message box. You will learn about other kinds of containers later. If no location–[*preposition container*]–is specified, **put** places *expression* in the Message box. Remember that a container holds a value and that the Message box is a container. You will learn more uses for **put** in subsequent chapters of this book.

Debugging and Testing

The distinction between *commands* and *functions* will become increasingly important as you work with HyperTalk. Keep in mind that a *function* produces a value that it either reports back to a command or puts into the Message box. A *command* immediately takes some action.

The Message box is an excellent debugging tool. If you create a script that doesn't seem to work right, you can try the statements from the script one at a time by typing them in the Message box. Keep in mind, however, that visual effects won't work when typed in the Message box.

If you are using a Macintosh keyboard with a numeric keypad, the Clear key can be used to erase all information from the Message box.

New HyperTalk Words in This Chapter

Commands
`show menuBar`
`show integer cards`
`show all cards`
`show objectName [at integer1,integer2]`
`show windowName [at integer1,integer2]`
`hide menuBar`
`hide objectName`
`hide windowName`

Functions
`the [adjective] date (or date())`
`the seconds (or seconds())`
`the ticks (or ticks())`
`the [adjective] time (or time())`

Containers
`[the] message [box]`
`[the] msg [box]`
`[the] message [window]`
`[the] msg [window]`

Object Modifiers
`next`
`prev[ious]`

Special
the

Prepositions
after
before
into

Window Names
card window
tool window
pattern window
[the] message (or msg) [window]
[the] message (or msg) [box]

Project Ideas

Create a card that provides "time" information. You might have date and time buttons. Perhaps you can even envision a use for information provided by **seconds** and **ticks**. Have each button put the appropriate information into the Message box.

Add a button to the first card of a stack that allows all of the cards to be displayed. You might want to put the "show all cards" icon on your button:

(This icon should be among the icons available with your version of *HyperCard*.)

Spend some time experimenting with the ideas in this chapter. Do you understand the syntax of all of the new commands and functions discussed? Do you understand the difference between a command and a function? Are you comfortable using the Message box?

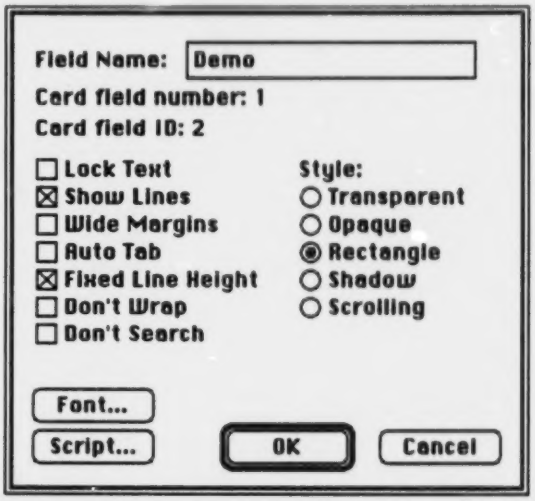
Create a card that has buttons on it to hide and show the Message box, the Tool window, and the Pattern window. You could use such a card whenever you are developing a new stack to place the windows you want to use at the locations of your choice.

If you didn't explore the screen coordinates by using **show** to place a window such as the Message box, carry out the exploration described in the "Show and Hide" section of this chapter.

Chapter 3. Working With Fields

In this chapter you will learn more about using the Message box with HyperTalk commands and more about writing scripts. You will gain further experience with **put**, **show**, and **hide**, learn about putting scripts on fields, and learn the new function **number**.

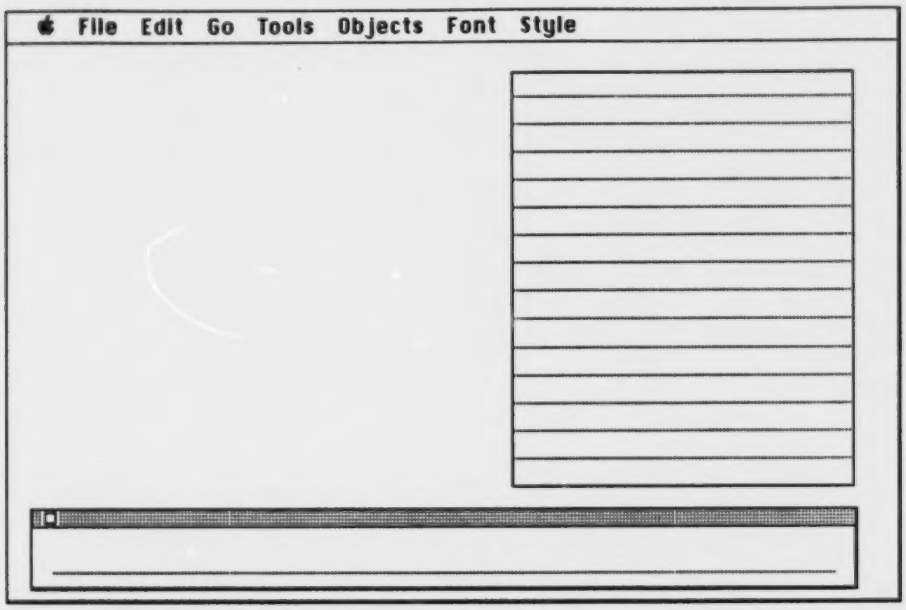
To begin, create a single field on a new card. Use **Field Info** under **Objects** on the Menu bar to get the appropriate dialog box:



Create a rectangular field with visible lines. Name the new field Demo.

Technical Note: If you are using a version of HyperCard *earlier than 2.0, this menu looks somewhat different.*

Resize the field and place it on the right side of the card. Display the Message box by using Command-M or by selecting **Message** from the **Go** menu:



Recall that the Message box is a container. You can put information into it. A field is also a container. (You will learn more about other types of containers in Chapter 23.) You know that you can type information directly into a field. Another way to put information into a field is to make use of the Message box.

Recall that using the Message box is a good way to explore using HyperTalk statements. To begin using HyperTalk to put text into fields, type

```
put "Hello" into card field "Demo"
```

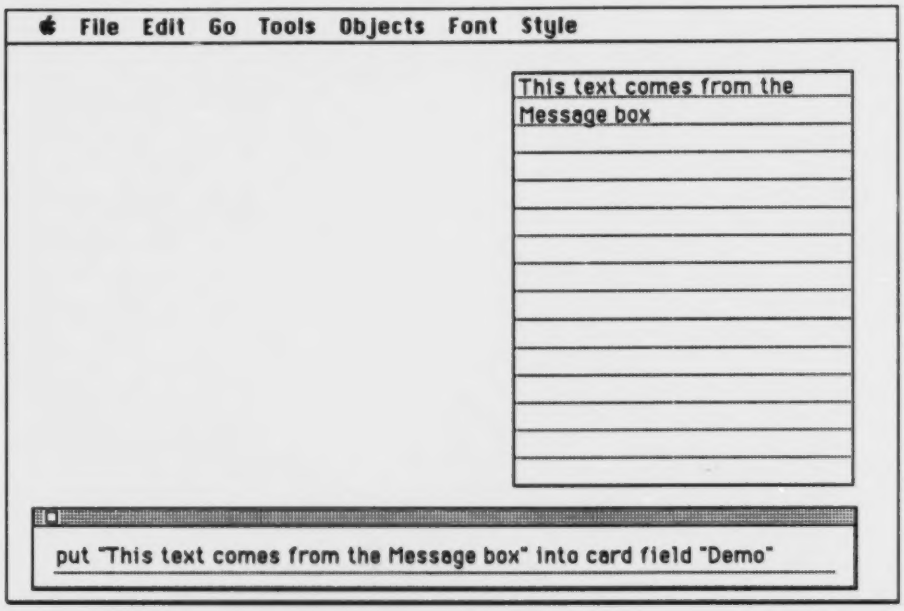
into the Message box and press Return. The word "Hello" appears on the first line of the field. Recall that **put** places information into a *container*, in this case, the field Demo. Note that it is necessary to use the words "card field" in the **put** statement to get "Hello" into the field rather than into the Message box.

Technical Note: You have learned that fields are objects *in* HyperCard. *Now you see that fields are also containers. Fields are unusual in* HyperCard *in that they can be thought of as either objects or containers.*

What happens if you put a longer string of text into the field? For example, what happens if you type

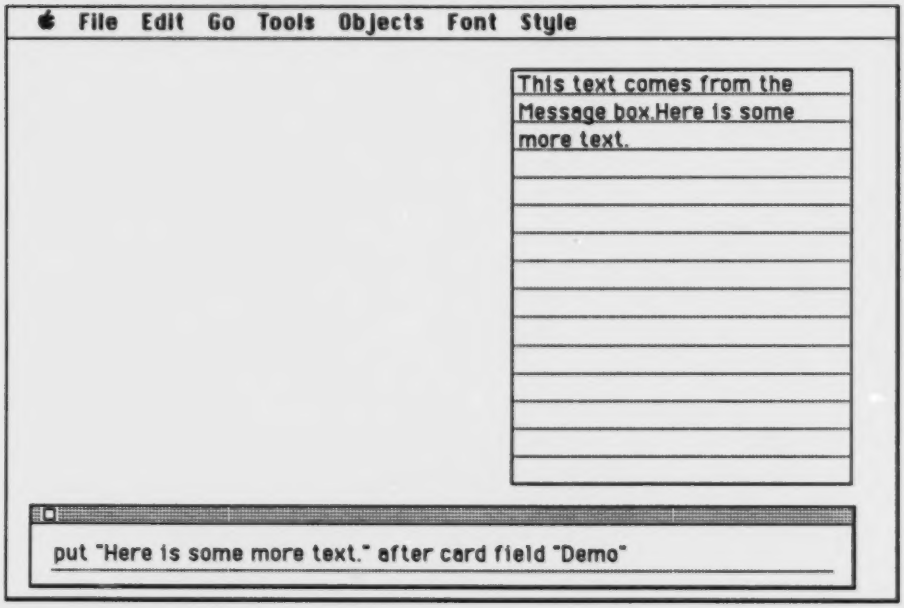
```
put "This text comes from the Message box" into card field
   "Demo"
```

You see that your text wraps to the next line:


File Edit Go Tools Objects Font Style
This text comes from the
Message box
put "This text comes from the Message box" into card field "Demo"

Did you note that this long line of text replaced the "Hello" that you put in the field earlier? You can add more text to text that is already in the field by typing

```
put "Here is some more text" after card field "Demo"
```


File Edit Go Tools Objects Font Style
This text comes from the
Message box.Here is some
more text.
put "Here is some more text." after card field "Demo"

In this case, the preposition "after" causes the new text to begin at the end of the text already in the field. You will learn more about working with text in fields and using prepositions such as "after" in Chapter 19 of this book.

Sometimes you want to be sure that there is no text in a field. You can type

```
put empty into card field "Demo"
```

"Empty" is a constant. It is used to represent "no characters." "Empty" is the same as "". Later you may want to use "empty" in a script to be sure there is no text in a particular field.

Using Buttons to Work With Fields

In the previous section, you saw that you can put text into fields by using the Message box. You can also put text into fields using scripts. To begin, create a button called "Experiment." Put the following script on the button:

```
on mouseUp
   put "Buttons can be used to put text into fields" into card
      field "Demo"
end mouseUp
```

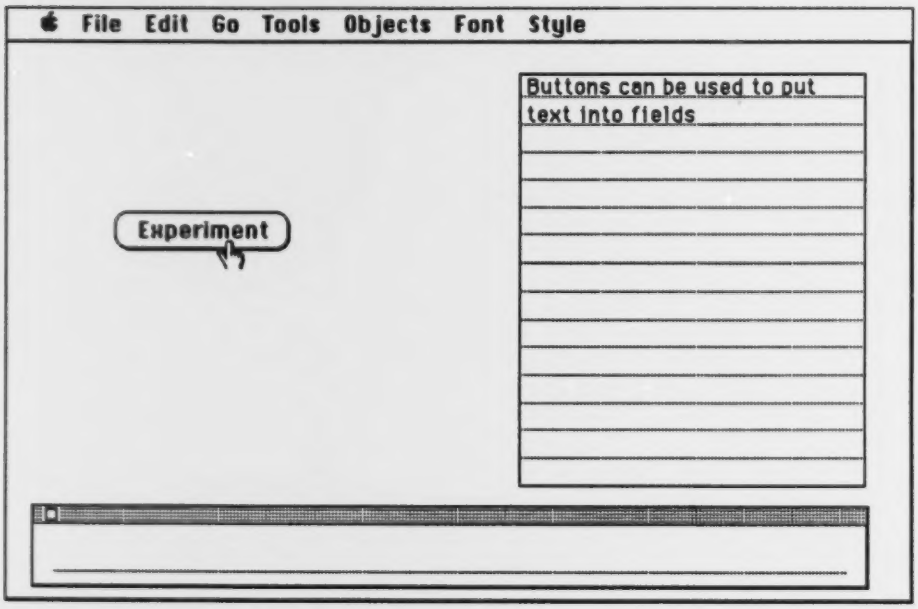
Did the "put" line get so long that it went off the right side of the Script window? You can put the continuation symbol (¬) nearly anywhere in the line to make your script more readable. (Press Option-Return to get the ¬ symbol.) For example you might type

```
on mouseUp
   put "Buttons can be used to put text into fields" ¬
      into card field "Demo"
end mouseUp
```

Note that you cannot put the ¬ symbol between two quotation marks. Try to break your lines before or after a group of quoted words.

Technical Note: If you split a group of words surrounded by quotations marks using the ¬ symbol, you will get an error message. In Chapter 17, you will learn how to use concatenation (&) to combine groups of words and avoid this problem.

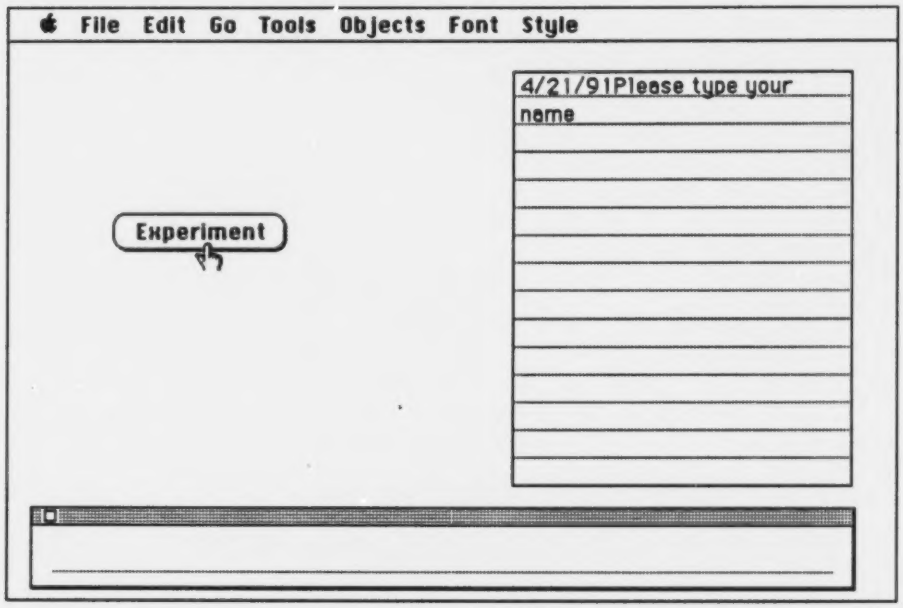
If you now click on the button you see



A more useful button script might be one that makes sure that the field is visible, and then provides the date and asks for the user's name. Try putting this script on card button "Experiment":

```
on mouseUp
   show card field "Demo"
   put the date into card field "Demo"
   put "Please type your name" after card field "Demo"
end mouseUp
```

The result should look something like this:



You can improve the appearance of the text in the field by modifying it as follows:

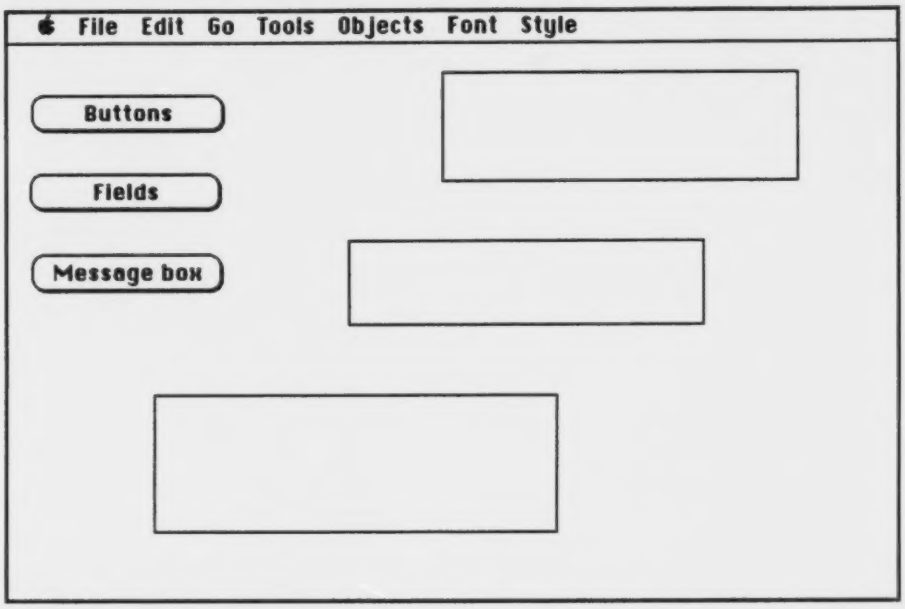
```
on mouseUp
   show card field "Demo"
   put the date into card field "Demo"
   put Return after card field "Demo"
   put "Please type your name" after card field "Demo"
end mouseUp
```

"Return" is a constant that represents the Return key (ASCII 13). This symbol is generally invisible and is used to end a line. Putting in a Return character causes "Please type your name" to go on the next line, exactly as if you had pressed the Return key after typing the first line.

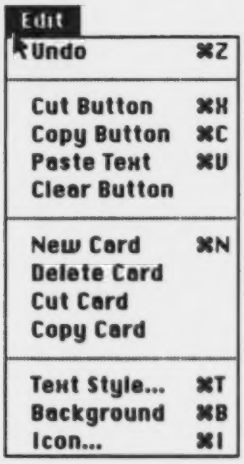
Scripting Fields

The previous example sets the stage for an "information" card containing both buttons and fields. Create a new card containing three buttons named "Buttons," "Fields," and "Message box," respectively. Also create three fields called "About

buttons," "About fields," and "About msg." The fields should be rectangular. Do not make them transparent. Your card might look like this:



Note that it is easier to create a number of similar buttons or fields by creating the first button or field in the style you want and then making copies. One way to copy a button or a field is to use the **Edit** menu (or the equivalent keyboard shortcuts) to copy and paste buttons. If you create one button and then click on that button, under the **Edit** menu you see



You can then use **Copy Button** (or Command-C) to make a copy of the button. When you pull the menu down again, you will see **Paste Button** (or you can use Command-V) and can paste a copy of the button. The copy of the button will be on top of the original button. Use the mouse pointer to move the copy. A similar process works for fields. Note that if you are not using *HyperCard* 2.0, this menu will look slightly different, but cutting, copying, and pasting work exactly the same way in other versions.

There is yet another shortcut for copying buttons or fields. After you choose either the Button tool or the Field tool, hold down the Option key. Continue to hold it down while you click on the button or field. Then, while holding down the mouse button, you can drag a copy of the button or field you have selected anywhere on the card.

On the button called Buttons, put this script:

```
on mouseUp
   show card field "About Buttons"
end mouseUp
```

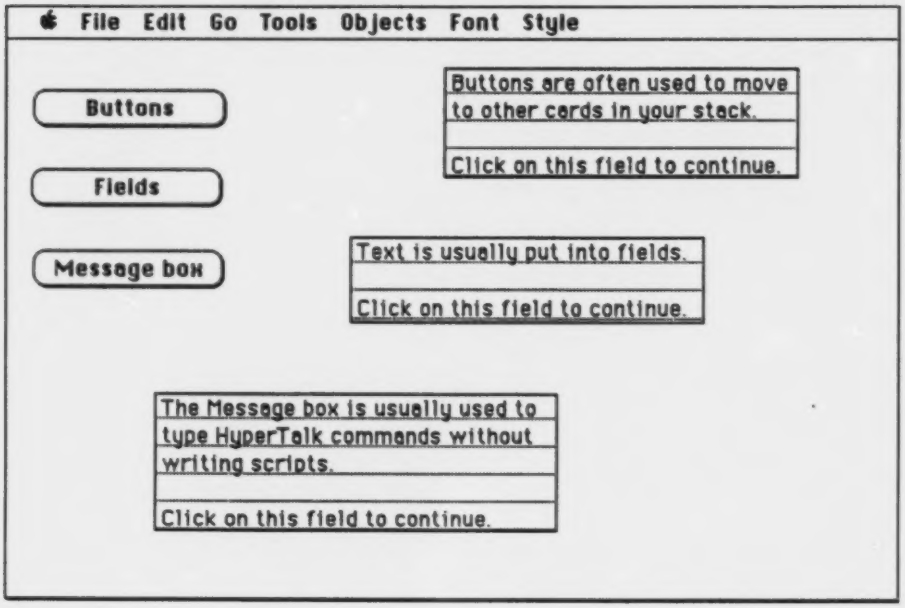
Put similar scripts on the Fields and Message box buttons:

```
on mouseUp
   show card field "About Fields"
end mouseUp
```

```
on mouseUp
   show card field "About Msg"
end mouseUp
```

If you can't see your fields even after clicking on the buttons, be sure your fields are not transparent. You can't see the outline of transparent fields even if you use **show** to display them.

Next, enter some text into each of the fields. Be sure to choose the Browse tool before you click in the field to begin typing.


File Edit Go Tools Objects Font Style
Buttons
Fields
Message box
Buttons are often used to move to other cards in your stack.
Click on this field to continue.
Text is usually put into fields.
Click on this field to continue.
The Message box is usually used to type HyperTalk commands without writing scripts.
Click on this field to continue.

When you have finished entering text, check the **Lock Text** box in the **Field Info** box under **Objects** for *each* of the three fields:

Finally, you need to put a script on each field. You can get to the script of a field in four ways:

- Choose the Field tool in the Tool window and select a field.

  Select the **Field Info** box found under the **Objects** menu.

  Click on the **Script** button.

- Choose the Field tool in the Tool window and select a field.

  Double click on the field

  Click on the **Script** button.

- Choose the Field tool in the Tool window.

  Hold down the Command and Option keys while clicking on the field.

- Choose the Browse tool in the Tool window.

  Hold down the Command, Option, and Shift keys while clicking on the field.

You can get to the scripts of buttons by using techniques similar to those just given. Note that you do not need the Shift key to get to the script of a button when the Browse tool is selected.

On the About buttons field, put

```
on mouseUp
   hide card field "About buttons"
end mouseUp
```

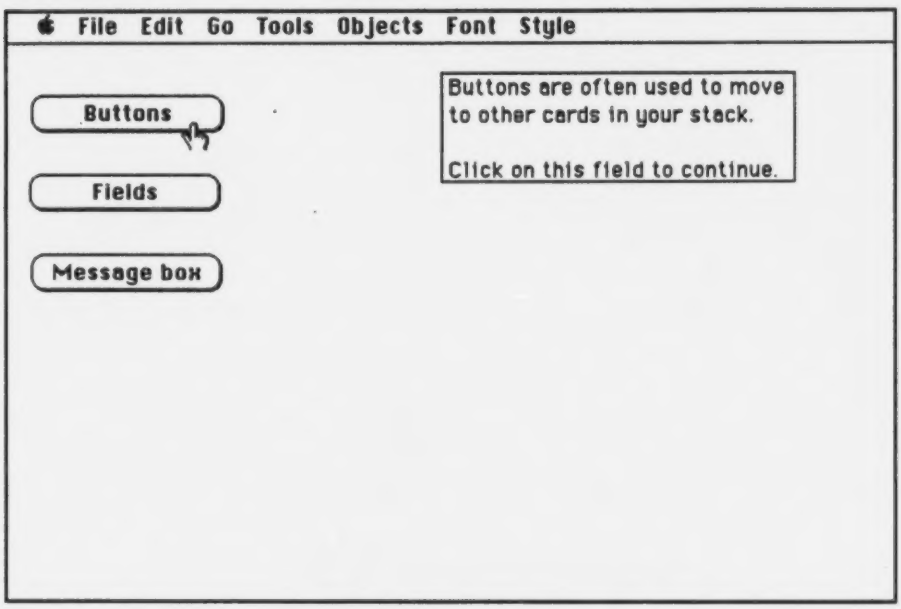
Put similar scripts on the other two buttons. On the field About fields put

```
on mouseUp
   hide card field "About fields"
end mouseUp
```

and on the About msg card field put

```
on mouseUp
   hide card field "About msg"
end mouseUp
```

Now, clicking on a field causes that field to disappear; clicking on a button causes the appropriate informational field to appear. This allows you to have as many fields as you want "open" at the same time. Such a card could be used to display "help" information, define terms, or give instructions to a student. Here is what your card might look like:


File Edit Go Tools Objects Font Style
Buttons
Fields
Message box
Buttons are often used to move to other cards in your stack.
Click on this field to continue.

Note that if your fields refuse to disappear as described, perhaps you forgot to lock them. Clicking on an unlocked field allows you to enter text in the field. A **mouseUp** handler on a field will work only if the field is locked.

As you can see, you can put a variety of kinds of scripts on fields just as you can put a variety of kinds of scripts on buttons. As you work with *HyperCard* you will gain a sense of what kinds of scripts are used on what kinds of objects. Scripting both with buttons and fields will be discussed in much more detail as you progress through this book.

Using Number

In the previous chapter you learned about functions. Recall that functions return a value. If you don't tell a function where to put a result, that result automatically appears in the Message box. The function **number** has a variety of uses. For example, you can find out how many buttons or fields are on a card by using the Message box instead of the pull-down menus, or you can count the number of characters in a field.

Using the information card from the previous example, type

```
the number of card buttons
```

into the Message box. The number "3" replaces the statement you typed in the Message box. Next, hide some or all of the fields. Then type

```
the number of card fields
```

Again, "3" appears even though some of the fields are hidden. This is a good technique for checking to be sure that you don't have "lost" objects!

You can also use **number** to count the "chunks" on the card—characters, words, items, and lines. Using chunk types, you can examine the text in a field by typing such commands as

```
the number of characters in card field "About Buttons"
```

or

```
the number of lines in card field "About Buttons"
```

in the Message box. The meaning of "characters," "words," and "lines" is fairly obvious. "Items" are really easy to understand as well. One item is separated from another by a comma. You will learn some more about chunk types and the ways in which they can be used to manipulate the text in fields in Chapter 19.

Did you notice that the word "character" and the word "line" were plural in the previous HyperTalk statements using **number**? Examining the syntax of **number** helps explain why. (Recall that *functions* are shown in a box surrounded with double lines.)

```
[the] number of objects
[the] number of chunkType in string
```

Here, *object* can be

```
[background] buttons
[card] fields
backgrounds
cards
```

while *chunkType* can be

```
characters (or chars)
words
items
lines
```

It is important that you pay careful attention to syntax descriptions of the HyperTalk commands and functions that you learn. However, you don't need to worry too much about *chunkTypes* just now. You will learn more about them in Chapter 19. However, do take note of the fact that *chunkTypes* are plural. Later you will see that if *chunk* is used in a syntax description, the singular nouns (character, word, item, line) must be used.

Did you also notice that "background" is optional when using **number** with buttons but "card" is optional when using **number** with fields? This means that

```
the number of buttons
```

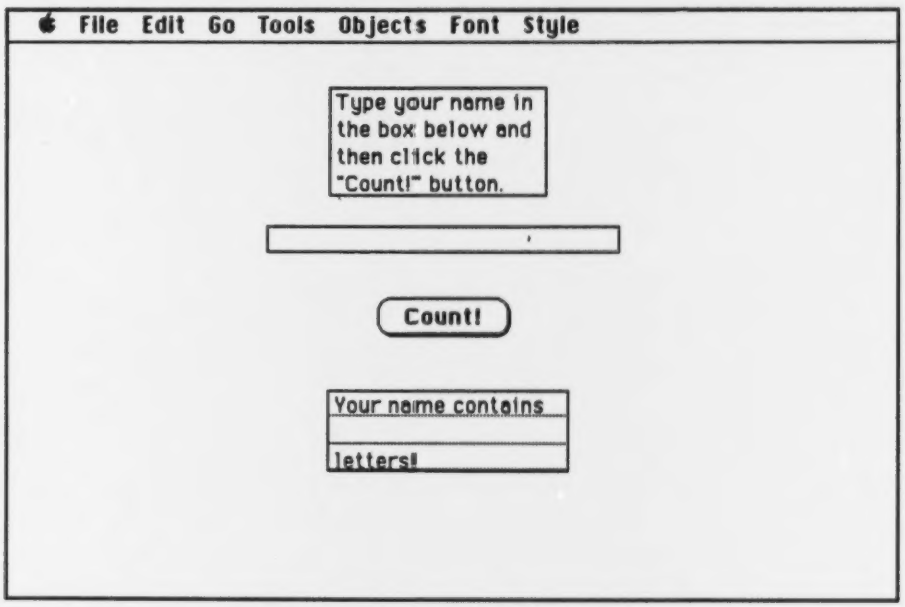
puts the number of *card* buttons in the Message box, while

```
the number of fields
```

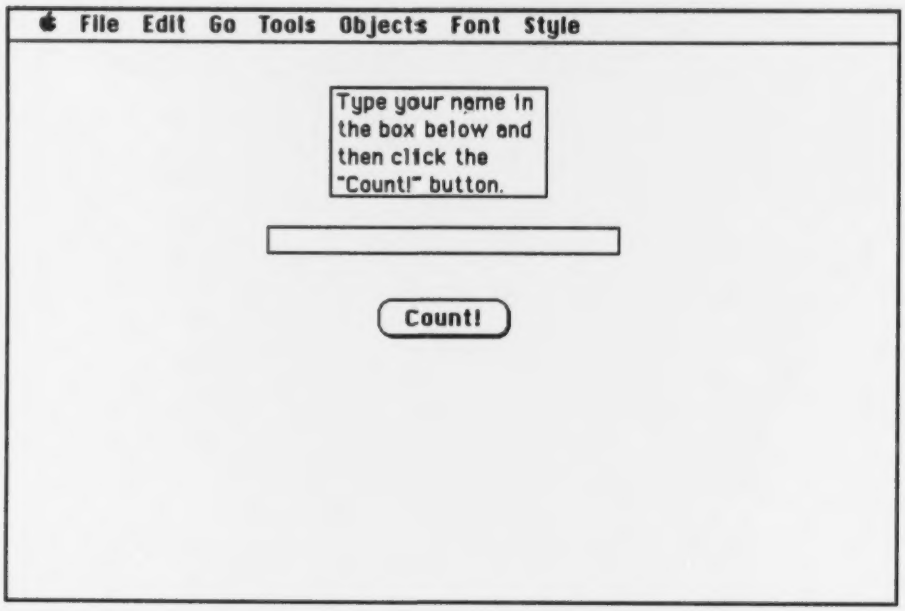
puts the number of *background* fields in the Message box. That is, buttons always default to the card while fields always default to the background. This detail can be hard to remember! A good rule of thumb when referring to any objects in HyperTalk is to

give the complete description. That is, *always* include such phrases as "card field" or "background button" in order to avoid confusion.

Next, practice using **number** and chunks by following this simple example. Create a card that looks like this:


File Edit Go Tools Objects Font Style
Type your name in the box below and then click the "Count!" button.
Count!
Your name contains
letters!

Hide the bottom card field so that the card looks like this:



Then, put a script such as this on the button:

```
on mouseUp
   show card field "Show count"
   put the number of characters in card field ¬
      "Name" into line 2 of card field "Show count"
   wait 3 seconds
   hide card field "Show count"
end mouseUp
```

Now when the user clicks in the Name field, types his or her name, and then clicks the Count! button, the number of characters in the user's name appears in the field Show count. Note that if you named your fields differently, your script will use your names instead of the ones shown in the previous illustration.

Debugging and Testing

It is important that you learn to use complete descriptions of objects when writing HyperTalk commands, that is, always say

```
card button "Demo"
```

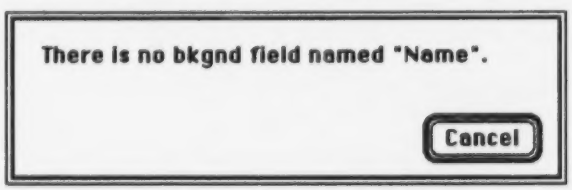
rather than

```
button "Demo"
```

Using complete descriptors will save you a lot of debugging time. For example, suppose you create a *card field* "Name" and want to put the word "Hello" into the field. If you type

```
put "Hello" into field "Name"
```

in the Message box, you might see the following dialog box:



HyperCard assumes that when you refer to a field, that field is in the background. But *your* field was on the *card*. You must include the word "card" to refer to a card field. It is easier to type full object descriptors than to remember a lot of rules such as this.

When you are checking your handlers, sometimes it is difficult to tell if the appropriate ones are working. One way to check is to use the **beep** command. You could put a different number of beeps in each of several handlers in a button or a field. For example, you might include

```
beep 1
```

in your **mouseUp** handler on a button and

```
beep 2
```

in the handler on a field. Using a technique such as this becomes increasingly helpful as you create more complex stacks.

The command **put** can also be useful when you are debugging. You can type such commands as

```
put the number of card buttons
```

or

```
put the name of card button 1
```

into the Message box in order to find out information about your card or a particular button. Sometimes it is faster to type these commands than to use the pull-down menus.

There are times when you have very long quoted lines you would like to break to make your scripts more readable. You can do this by using the & (concatenation) symbol. For example:

```
on mouseUp
   put "This is a very long line of text that I want to " & ¬
      "put into a field." into card field 1
end mouseUp
```

Note that there is a space after the last word in the first line of the quoted list. If you omit the space, the word "to" and the word "put" will have no space between them when the line is put into the field. Next comes the concatenation symbol, and finally the continuation symbol to tell HyperTalk that the rest of the line of code is on the next physical line.

New HyperTalk Words in This Chapter

Chunk Types
characters (or chars)
words
items
lines

Constants
empty
return

Prepositions
after

Functions
the number or number()

Symbols	
Option-Return	continuation symbol (¬)
Command key	⌘

Shortcuts	
Command-Option drag	to copy a selected button or field
Command-Option-click	to get to the script of a button or field when the Button or Field tool is selected
Command-Option-click	to get to the script of a button when the Browse tool is selected
Shift-Command-Option-click	to get to the script of a field when the Browse tool is selected

Project Ideas

Spend some time working with fields from the Message box. Can you put information into fields? Can you show and hide fields? Can you tell how many fields are on the card or on the background?

Spend some time reviewing scripting buttons. You might practice using buttons to open fields, put information into fields, or make fields visible or invisible.

Experiment with the **number** function from the Message box. Try using it to count words or lines in a field.

Design some buttons to provide the user with information about how many buttons and fields are on a card.

Chapter 4. Introduction to Button Properties

In this chapter you will learn about *HyperCard* properties. In particular, you will explore the properties associated with buttons. You will learn how to use HyperTalk to work with button properties.

In your initial work with *HyperCard*, you learned that it allows you to work with a number of *objects*: stacks, cards, backgrounds, fields, and buttons. Each of these objects has a number of characteristics, such as size, shape, and position. These characteristics are called *properties*. Think for a moment about the dialog box you see when you select **Button Info** from the **Objects** menu. You might, for example, see

You can see that this button has the name "Experiment." It has a number of "1" and an ID of "3." (Recall that the ID of a button is unique; it will never change or be duplicated elsewhere in the stack, while the number of a button may change and other objects may have the same number.) Its name will appear on the button because **Show Name** is checked. When you click on the button, it will be highlighted because **Auto Hilite** is checked, and its shape will be a round rectangle because **Round Rect** is checked. Each choice in a dialog box such as this represents a *property*.

Getting Started Using HyperTalk With Button Properties

The HyperTalk words for representing the properties of buttons that appear in the **Button Info** dialog box are

```
name
number
ID
showName
autoHilite
icon
script
style
```

Whenever you have created a button, you probably set some of these properties using the dialog box. You have seen that properties such as **name**, **number**, and **ID** have words or numbers associated with them. Properties such as **autoHilite** and **showName** are either checked (true) or unchecked (false). Some of the button properties in the dialog box can only be examined, while others can be set. Similarly, some button properties can be set using HyperTalk, while others can only be examined.

To begin exploring the use of HyperTalk with button properties, start with a card called "Demo" and a button called "Experiment." If you type

```
the name of card button 1
```

in the Message box, you see

```
card button "Experiment"
```

Technical Note: If you are working with object or window properties (which are discussed in Chapter 8), you must *precede the property name with "the" or follow it with "of." However, "the" is not necessary with button and field properties. However, in this book, "the" will be used before the word "property" for consistency and to make HyperTalk code more readable.*

The **name** property has three possible *adjectives* that can be used to modify it. They are **short**, **abbreviated**, and **long**. If you type

```
the short name of card button "Experiment"
```

you see

```
Experiment
```

If you type

```
the abbreviated name of card button "Experiment"
```

you see

```
card button "Experiment"
```

Note that **abbreviated name** and **name** give the same result. Also notice that the **short name** does not include quotation marks since no words other than the name of the button itself are given. On the other hand, the **name** (or **abbreviated name**) does include quotation marks.

If you type

```
the long name of card button "Experiment"
```

you will get the full *path name*, which might look something like this:

```
card button "Experiment" of card "Demo" of stack
   "MyHardDrive:HyperCard:My Experiments"
```

The path name describes the location of your stack. It begins with the name of the disk you are using, followed by the name of the folder or folders in which your stack is located. If the stack you are working with is inside several folders, only part of the long name will be visible in the Message box. However, if you use the mouse to select and delete the characters that are visible, the rest of the path name will appear.

Using Set

By using the HyperTalk command **set**, you can change the name of a button. **Set** is a new command. Its syntax is:

```
set [the] property [of object] to string
```

Here *property* is one of the many *HyperCard* properties and *object* is an object descriptor or window name. A *string* is any collection of characters—letters, numbers, and/or symbols. Here *string* must be a legal value for the property being set.

To change the name of the Experiment button, you might type

```
set the name of card button "Experiment" to "New Experiment"
```

The name "New Experiment" appears on your button. (Change the name back to "Experiment" to continue the explorations in this chapter.)

The **number** property and the **ID** properties of a button can't be changed using the **set** command, but you can see the values of these properties by using the Message box. For example, you can type

```
the number of card button "Experiment"
```

or

```
the ID of card button 1
```

Sometimes you will find that the **number** and **ID** of the card button are the same. This will not generally be the case. The **number** of a card is set by *HyperCard* and may change as you add, delete, or sort cards. The **ID** of a button is a permanent number assigned by *HyperCard*. It will not change as you work with your stack.

Clicking on the **Show Name** check box in the **Button Info** dialog box sets the **showName** property to "true." Similarly, checking **Auto Hilite** sets **autoHilite** to "true." Each of these properties can have the values of the constants "true" or "false" and can be changed by using the **set** command. Thus, you can type the following lines in the Message box one at a time:

```
set the showName of card button "Experiment" to false
set the autoHilite of card button "Experiment" to false
```

You should now see a blank button, which, if you click on it, will not become highlighted.

A button may have an icon on it. Each icon has a name and number associated with it. The **icon** property of the button allows you to find out about the icon appearing on the button. If you type

```
the icon of card button "Experiment"
```

the number 0 appears in the Message box since this button does not have an icon on it. The **icon** property has the value of zero when there is no icon on the button.

You have probably put icons on your buttons in your earlier work with *HyperCard*. To understand the **icon** property better, go to the **Button Info** dialog box and click on the **Icon** button. You see a screen that looks something like this:

Notice that when you select an icon, you see an icon ID number and a name at the top of the screen. Note also that this dialog box may look somewhat different if you are using a version of *HyperCard* earlier than 2.0.

You can use either the icon ID number or the icon name to put an icon on a button when using the **set** command. For example, you might put the icon highlighted in the previous figure on the Experiment button by typing

```
set the icon of card button "Experiment" to 1012
```

or

```
set the icon of card button "Experiment" to "Return Arrow"
```

Either of these commands will put the Return Arrow icon on the Experiment button. You can remove an icon from the button by typing

```
set the icon of card button "Experiment" to 0
```

since the **icon** property is zero when the button does not have an icon.

Using button properties such as these, you can write scripts that cause buttons to change. For example, you might put the following script on the Experiment button:

```
on mouseUp
   set the name of card button "Experiment" to "Surprise"
end mouseUp
```

When you click on the button, the name changes! (Change the name of the button back to Experiment to continue the explorations in this chapter.)

The **script** property represents the *entire* script on the button. Thus, if you type

```
set the script of card button "Experiment" to empty
```

all of the handlers that appear in the script editor are replaced. (Recall that "empty" is a constant representing a string of no characters.) You can use the **script** property to retrieve a script if you wish. For example, suppose the Experiment button has the following handler:

```
on mouseUp
   beep 5
end mouseUp
```

If you type

```
the script of card button "Demo"
```

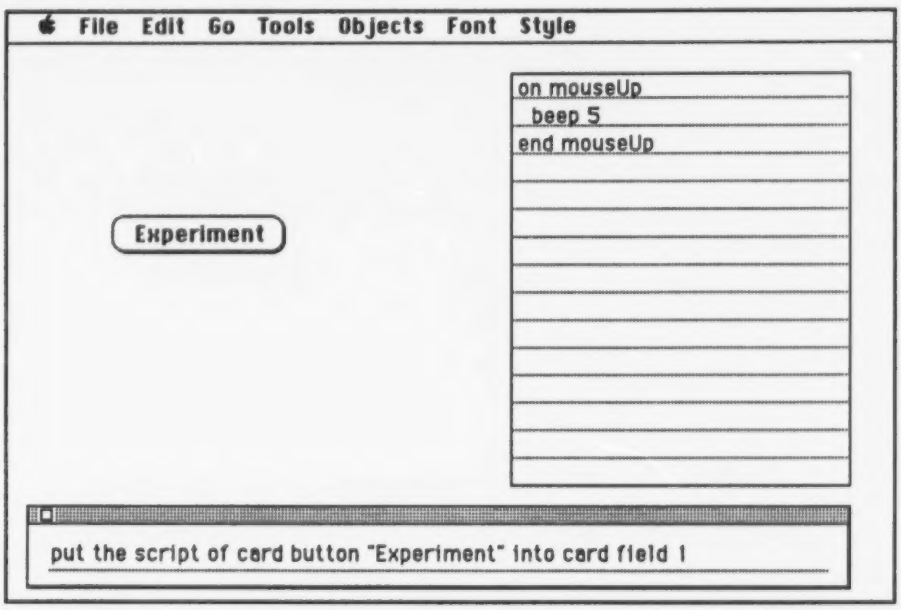
you see

```
on mouseUp
```

in the Message box. Recall that only one line (ending with a carriage return) can be in the Message box at one time. If you have (or create) a field on your card Demo, you can write

```
put the script of card button "Demo" into card field 1
```

and you will see something like this:


File Edit Go Tools Objects Font Style
on mouseUp
beep 5
end mouseUp
Experiment
put the script of card button "Experiment" into card field 1

On the right side of the **Button Info** dialog box is a list of possible button styles. This list represents the possible values for **style** property. In HyperTalk they are:

```
transparent
opaque
rectangle
roundRect
checkBox
radioButton
```

Like other properties, you can find out the current style of a button by typing something like

```
the style of card button "Experiment"
```

into the Message box. You can change the style by using something like

```
set the style of card button "Experiment" to transparent
```

You will see the style of the button change. Try it!

Yet More Properties

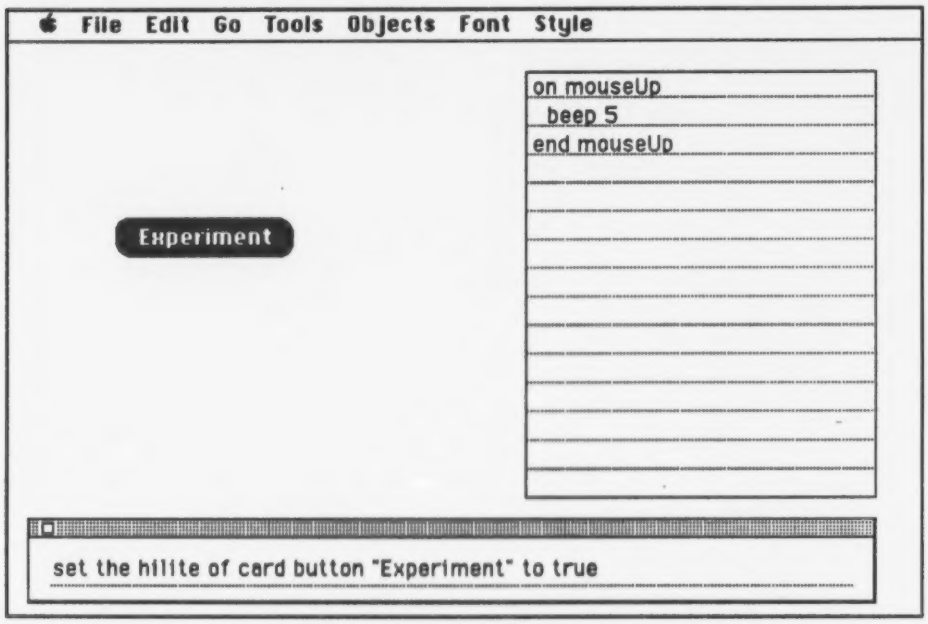
In addition to the properties of buttons that you can change using the **Button Info** dialog box, there are button properties that you can only change using HyperTalk. One example of such a property is the

```
hilite
```

property. When you create a button, the **hilite** property is automatically set to "false." If you type

```
set the hilite of card button "Experiment" to true
```

in the Message box, the button turns to black:



Note that when **hilite** is set to "true" the button is black (inverse), and when **hilite** is "false" the button is white. **Hilite** is different from **autoHilite**. When **autoHilite** is set to "true" the button becomes black during the time you are holding the mouse button down. When you release the button, it immediately becomes white again. When **autoHilite** is "false," the button does not turn black at all when you click on it.

When you change the position of a button on the screen, you are changing its **location** property. The

```
loc[ation]
```

property indicates where the button is found on the screen. If you type

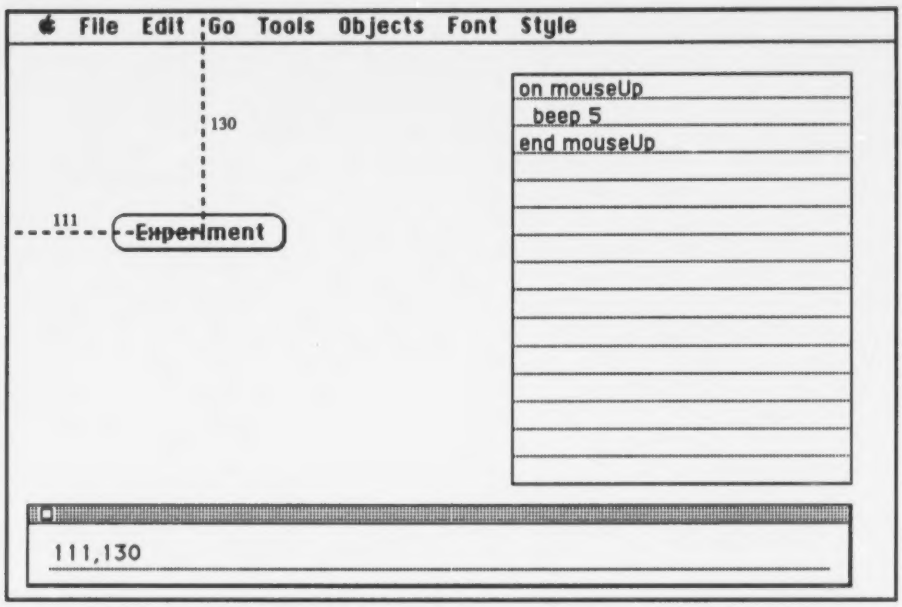
```
the location of card button "Experiment"
```

in the Message box, you see something like

```
113,130
```

These numbers represent the coordinates of the center of the button. The Macintosh screen is made up of tiny dots, called pixels. The coordinates give the number of pixels from the top left corner of the card to the center of the button. The first coordinate gives

the horizontal distance from the corner; the second coordinate gives the vertical distance from the corner:



You can use **set** to change the location of the button on the screen. For example, you could put this handler on the Experiment button:

```
on mouseUp
   set the location of card button "Experiment" to 250,200
end mouseUp
```

Now, clicking on the button causes it to move to the right and down on the card. You might use this "moving button" to draw attention to something on your card!

When you change the size of a button by clicking and dragging, you are changing a property of the button. This property is the

```
rect[angle]
```

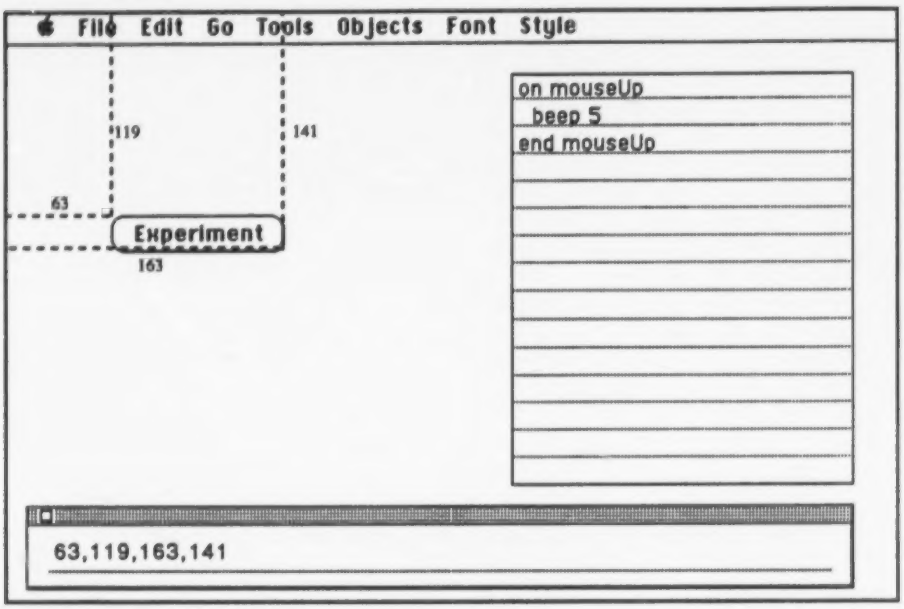
property. The value of **rectangle** is four numbers. The first two numbers are the coordinates of the top left corner of the button. The second two numbers are the coordinates of the bottom right corner of the button. If you type

```
the rectangle of card button "Experiment"
```

in the Message box, you see something like

```
63,119,163,141
```

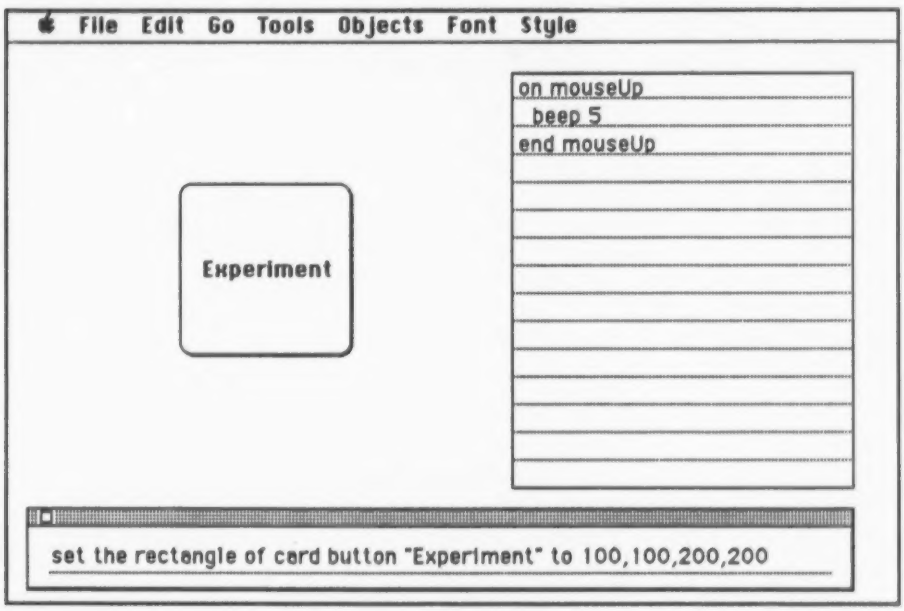
Here, 63 and 119 represent the top left corner of the button, while 163 and 141 represent the lower right corner of the button:



Changing the **rectangle** property changes the size of the button. For example, you might type

```
set the rectangle of card button "Experiment" to
   100,100,200,200
```

in the Message box. The size of the button then changes:



You might use a button that changes size to get a special effect, for example:

```
on mouseUp
   set the rectangle of card button "Surprise" to
      100,100,200,200
   wait 15 ticks
   set the rectangle of card button "Surprise" to 95,95,205,205
   wait 15 ticks
   set the rectangle of card button "Surprise" to
      100,100,200,200
end mouseUp
```

When you click on this button, it gets a little larger and then goes back to its original size.

Debugging and Testing

When using the **script** property, you need to be especially careful that you don't accidentally wipe out entire scripts. Remember that *all of the text* in the script editor is removed and replaced when you use

```
set the script of ...
```

Note that you use "into" with **put** and "to" with **set**; that is, you write

```
put the name of card button "Surprise" into the Message box
```

but you write

```
set the visible of card button "Surprise" to true
```

When you place a *value* into a *container* by using **put**, you use the preposition "into." When you want to change the value of a *property*, you use the preposition "to" with **set**.

In order for an icon to be used on a button, it must be available to your stack. If you have problems with icons not appearing as expected, you should check the resources by clicking on the **Icon** button in the **Button Info** dialog box.

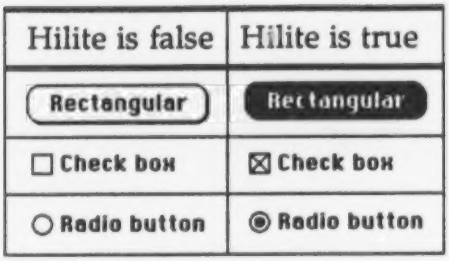
If you type something like

```
the name of card button 1
```

in the Message box and the card does not have a name, you see the ID of the card instead of the name.

Keep in mind that the **hilite** property is automatically set to "false" when you create a button. In general, **hilite** remains "false" unless you set it to "true." In the case where **autoHilite** is set to "true," clicking on the button causes it to turn black when you hold down the mouse button. During the time that the button is black, **hilite** becomes "true." When you release the mouse button, **hilite** is reset to "false." You need to keep in mind that the value of the **hilite** property can change when **autoHilite** is "true."

You can highlight any style of button, but different buttons show this highlighting in different ways. The rectangular-type buttons invert when they are highlighted. Check boxes display an "X," and radio buttons show a little circle. The following chart shows different types of buttons in unhilighted and highlighted form.



Using the **location** property, it is possible to set a button's location off the card. For example, if you type

```
set the location of card button 1 to 500, 100
```

the button moves part way off the edge of a standard size *HyperCard* card. If you type

```
set the location of card button 1 to 600, 100
```

the button disappears. Clearly, a typographical error can result in a "lost" button. You can always use **set** to move a lost button to a location on the card where you can see it.

You can also place buttons off the card by using the **rectangle** property. In addition, you can make buttons that are smaller than the "minimum" size. Thus, if you type

```
set the rectangle of card button 1 to 100,100,105,105
```

you create a button 5 dots (pixels) on a side. *HyperCard* will let you use such a tiny button. However, if you try to modify this button by using the Button tool, *HyperCard* will resize it. The minimum size of button you can edit with the mouse is 12 pixels on a side. In addition, if you type

```
set the rectangle of card button 1 to 100,100,90,90
```

you have created an invisible button. Again, a typographical error can result in lost buttons. You can, of course, see this button again by typing something like

```
set the rectangle of card button 1 to 100,100,120,120
```

You can type statements like

```
the number of card buttons
```

in the Message box to find out how many buttons are on the card. You can then use statements like

```
the name of card button 1
the location of card button 1
```

to locate the buttons in order to change or remove them.

When working with the **rectangle** and **location** properties, it is sometimes helpful to be able to determine the coordinates of a point on the screen before setting these properties. Here's a technique that can save you time. First, put your mouse at the spot on the card for which you want coordinates. *Without touching the mouse,* type

```
the mouseLoc
```

The characters you type will appear in the Message box. Then the coordinates of the mouse pointer appear in the Message box. This technique allows you to get the location of any tool that has been selected in the Tool window.

Technical Note: Many functions have two forms. One form requires that you use the word "the" in front of the function name. The second omits "the" but includes parentheses after the function name. Thus, you can write either the ***mouseLoc*** *or* ***mouseLoc()****.*

New HyperTalk Words in This Chapter

Properties
name
number
ID
showName
autoHilite
icon
script
style
location
rectangle

Button Styles
transparent
opaque
rectangle
roundRect
checkBox
radioButton

Adjectives
abbrev[iated]
long
short

Constants
true
false
false

Project Ideas

Create a new button by using the Button tool. Instead of using the **Button Info** dialog box, use HyperTalk commands to set all the properties of the button. Experiment with different properties. Be sure you know how to use each one.

Create a "Button creator" card. Your card should have a series of meaningfully named buttons and a generic "New Button." Clicking on one of the named buttons should change the new button to the appropriate form. You could then copy this newly created button to another card. Thus, you might have a "Square Button" button with a script such as

```
on mouseUp
   set the showName of card button "New Button" to true
   set the hilite of card button "New Button" to true
   set the autoHilite of card button "New Button" to false
   set the style of card button "New Button" to rectangle
   set the rectangle of card button "New Button" to
      200,200,250,250
   set the name of card button "New Button" to "Square"
end mouseUp
```

How many other button types can you create? Explore. Have fun!

Use some of the properties in this chapter to create special effects. For example, clicking on a "Look!" button might cause the button to move across the screen to draw attention to a picture. Or perhaps you can make use of the **icon** property so that clicking on a button causes the picture on the button to change. This might be the basis for some kind of game. What other kinds of special effects can you create by using button properties?

Chapter 5. The Play Command

In this chapter you will learn how to make musical sounds using the HyperTalk command **play**.

The sounds you have made so far in your work with *HyperCard* have only been created using the **beep** command. **Beep** uses the sound that has been selected on the Control Panel under the **Apple** (🍎) menu. This is usually the "Simple Beep," but other sounds can be substituted. The **play** command allows you to play musical scales with different types of sounds. *HyperCard* includes the sounds "Boing" and "Harpsichord." (If you are using *HyperCard* 2.0, you also have the "Flute" sound.)

Technical Note: You can also use digitized sound and voice samples that are part of the resources in your stack. Check a technical reference source, such as Inside Macintosh, *Volume V, for more information on sound resources.*

Try typing

```
play "boing" "c d e"
```

in the Message box. You hear three "boings" going up the scale. Next, try typing

```
play "boing" tempo 100 "c d e"
```

You hear the same three notes, perhaps at a different speed. Now change the 100 to 1000:

```
play "boing" tempo 1000 "c d e"
```

Now you hear the notes at a much faster speed. You can now see that the **play** command has three parts. The syntax is:

```
play "string1" [tempo number] "string2"
```

The first parameter, "string1," is the name of the sound. This may be one of the built-in sounds, or it may be a digitized sound. To avoid confusion, you should always put quotations marks around the name of the sound. Using "tempo" is optional; if "tempo" is omitted, *HyperCard* defaults to 120. "Tempo" has one parameter—a number representing the speed with which the sounds play. "*String2*," the third parameter, is the list of notes to play. It must contain fewer than 256 characters. Note that it is a good idea to put quotation marks around this list as well.

Representing Notes

To begin exploring the **play** command, type

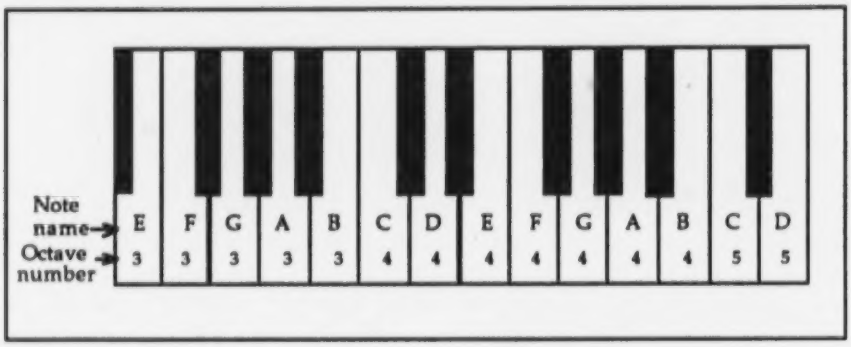
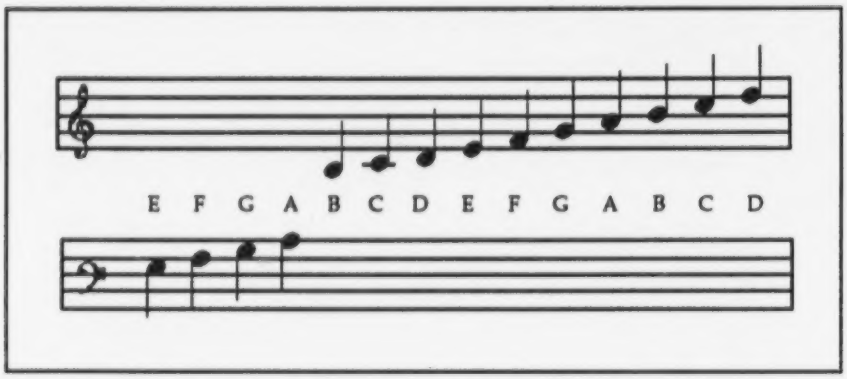
```
play "harpsichord" "c4q c#4q d4q d#4q e4q f4q f#4q g4q"
```

You hear a scale. "C4q" represents a single note. "D#4q" represents another note. There are four possible parts to each note representation. First comes the note name. Notes are named using the conventional "A B C D E F G" notation. An "R" or a zero is used to represent a rest.

After the note name comes the *accidental*. An accidental is either a sharp (represented by #) or a flat (represented by b). Otherwise the note is natural

(represented by nothing.) Thus, C#4q represents the note C sharp, while Cb4q represents the note C flat. C4q represents C natural.

Next comes the octave number. Higher numbers give higher pitches. The number "2" is probably the smallest number you will want to use to get musical sounds. If you do not include an octave number, *HyperCard* defaults to octave 4, the octave containing middle C.

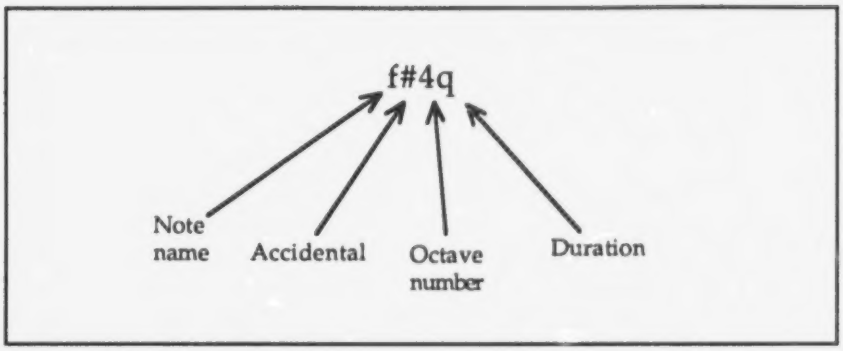


Seven is probably the highest octave number you will want to use. If you use higher numbers, the pitches you hear will not be what you expect.

The final part of a note name is the *duration*. Duration is represented by

```
w  whole note
h  half note
q  quarter note
e  eighth note
s  sixteenth note
t  thirty-second note
x  sixty-fourth note
```

In addition, you can use a period after the duration to specify a dotted note (1 1/2 times the normal duration) or put the digit 3 after the duration to represent a triplet note (2/3 the normal duration). The longest possible note duration is slightly more than 16 seconds.



Note that if you do not specify a duration, *HyperCard* defaults to a quarter note.

Using the Play Command

There are many possible variations when using the **play** command. If you examine the syntax of **play** carefully, you notice that the tempo is optional. In addition, you do not have to specify the octave number and the duration for each note. Look again at the example given at the beginning of this chapter. Each note had the octave and duration specified. Once you specify an octave, the notes that follow will stay in the same octave until you change it. Thus,

```
play "harpsichord" "c4q c#4q d4q d#4q e4q f4q f#4q g4q"
```

can be written

```
play "harpsichord" "c4q c#q dq d#q eq fq f#q gq"
```

since all of the notes are in the fourth octave, the octave containing middle C. Changing the octave to 5 midway in the list will make all the notes beginning with the change play in the next higher octave. For example, if you type

```
play "harpsichord" "c4q c#q dq d#5q eq fq f# gq"
```

d# and all of the notes after it will play in the fifth octave. Duration also can be carried to subsequent notes. Thus, you can write

```
play "harpsichord" "c4q c# d d# e f f# g"
```

and hear only quarter notes. If you change the d# to d#h,

```
play "harpsichord" "c4q c# d d#h e f f# g"
```

all of the notes beginning with d# will be played as half notes.

The duration given with each note or group of notes tells *HyperCard* how fast to play that note or group of notes. When you include tempo in the **play** command, you change the speed at which all of the notes are played. You saw earlier that if you type

```
play "harpsichord" tempo 100 "c4q c# d d# e f f# g"
```

you hear a slow scale, while if you type

```
play "harpsichord" tempo 1000 "c4q c# d d# e f f# g"
```

you hear a very rapid scale. Recall that if no value is given for tempo, *HyperCard* uses whatever value was last used with tempo.

Suppose you create a button with music in the script, for example

```
on mouseUp
   play "harpsichord" tempo 100 "c4q c# d d# e f f# g"
   go to next card
end mouseUp
```

If you click on this button, the music continues to play even after the next card is visible. Music continues to play while other actions occur in *HyperCard*. You can control what happens with music by using

```
play stop
```

Modify the previous script as follows:

```
on mouseUp
   play "harpsichord" tempo 100 "c4q c# d d# e f f# g"
   visual effect dissolve slowly
   go to next card
   play stop
   go to previous card
end mouseUp
```

Mostly likely you will hear the music stop in the middle of the scale. After *HyperCard* encounters the **play stop** command, the music stops.

Technical Note: The number of notes you hear depends on the speed of the Macintosh you are using. The speed even depends on whether you are using a floppy disk drive or a hard drive. If you hear all of the notes, remove the "slowly" from the handler in the script.

Of course, the **play** command can be used to add little tunes to your stack or to play recorded sounds. However, you can also use **play** to add auditory prompts and cues to your stack. For example, you might put the following script on a button:

```
on mouseUp
   play "harpsichord" tempo 100 "g6s cs"
   go to next card
end mouseUp
```

This script creates a gentle, high-pitched sound when you click on the button. Such an auditory cue makes it clear to users that they have actually clicked on the button.

Debugging and Testing

It is not uncommon for *HyperCard* users to decide that they can't work with sounds because they are not "musical." You don't have to know much about music to make use of the **play** command. If you borrow some beginning band music in the key of C and make use of the charts at the beginning of this chapter, you should be able to create *HyperCard* music with relative ease.

If you put a name of a sound that does not exist after **play**, you will hear nothing at all. You also need to be sure that the sound on your Macintosh is not turned off. If the sound is off, the Menu bar flashes when you would normally hear a "beep."

Be particularly careful about typing **play** commands. A misspelling of the sound name or a typographical error in some other part of the command can have unexpected results. Sometimes your music won't play correctly. Sometimes you will hear extra notes, or no notes at all. Similar problems can occur if you don't use quotation marks around the sound name and around the list of notes.

Note names can also be represented as numbers. Middle C is 60; the B below middle C is 59; the D above middle C is 61. Using numbers can be helpful in your future work with *HyperCard*, and this is one of the reasons it is important to be particularly careful about spacing and punctuation in **play** commands.

New HyperTalk Words in This Chapter

Commands
`play`
`play stop`

Words and Symbols Used With Play
`a b c d e f g r with # or b (for musical notes)`
`w h q e s t x with . or 3 (for note durations)`
`tempo`

Project Ideas

Even if you think you are not "musical," try writing some *HyperCard* music anyway! All you need to do is get some beginning instrumental music from the music department in your school or from a music store. If you are unfamiliar with traditional musical notation, you should get music in the key of C. This will avoid problems with those "accidentals" mentioned earlier in this chapter. Use the keyboard chart given in this chapter to help you get started.

Here is a note list for the first part of "Three Blind Mice":

```
eq4 d ch eq d ch
```

Create several buttons that will play this song slowly, quickly, in a high range, in a low range, and so on.

If you have musical experience, create a card with a series of buttons for musical exploration. Each could play a different tune, a different phrase of a larger work, or even a different interval. Use your imagination to create a card (or stack) that would help teach something about music.

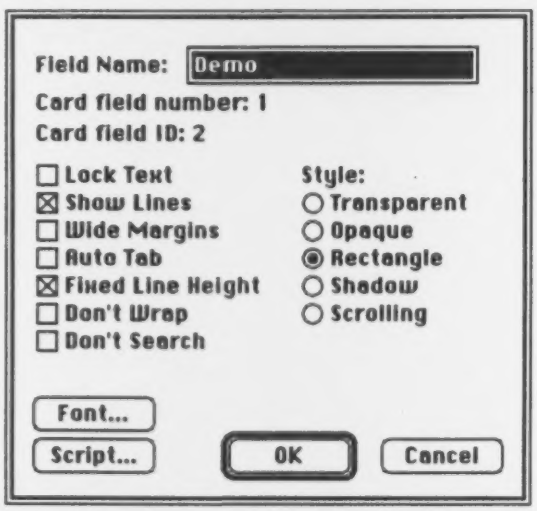
You can create interesting sound effects by playing notes in very low or very high ranges or by playing them very quickly. Experiment with some groups of notes and create a series of sound effects buttons.

Chapter 6. Using Field Properties

In this chapter, you will learn how to use HyperTalk to work with the field properties that can be set in the **Field Info** dialog box. You will also learn to use some field properties that are not available from the **Field Info** dialog box.

In Chapter 4 you learned that *HyperCard* objects have properties with which you can work by using HyperTalk. In particular, you examined the properties associated with buttons. Fields have a number of properties as well. Many of the properties are quite similar to the properties associated with buttons.

Recall that the **Field Info** dialog box looks like this:



Note that if you are using a version of *HyperCard* before version 2.0, there will be fewer choices in the **Field Info** dialog box.

Getting Started With Field Properties

In Chapter 4, you learned that there were HyperTalk names for all the properties that were in the **Button Info** dialog box. Similarly, there are HyperTalk names for the properties that you can set in the **Field Info** dialog box. These property names are:

```
name
number
ID
lockText
showLines
wideMargins
autoTab
font
script
style
```

If you are using *HyperCard* 2.0, you can also work with these properties:

```
fixedLineHeight
dontWrap
dontSearch
```

As you might guess, working with field properties by using HyperTalk commands is much like working with button properties by using HyperTalk. For example, you can determine the name of a field by typing

```
the name of card field 1
```

in the Message box. The name of the field appears. Recall that you can use **long**, **short**, and **abbreviated** as adjectives for **name** with buttons. You can also use these modifiers with the names of fields. As with buttons, you can **set** the name of the field by typing something like

```
set the name of card field 1 to "My Field"
```

Similarly, the **number** and **ID** of a field can be determined. If you type

```
the number of card field 1
```

into the Message box, the **number** of the first field on the card appears. If you type

```
the ID of card field 1
```

the **ID** of the first card field appears. Like buttons, neither **number** nor **ID** can be changed by using **set**.

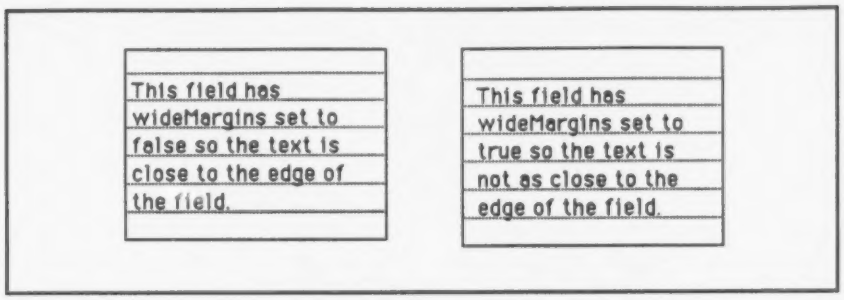
The **lockText** property determines whether or not you can type information into a field. If **lockText** is **true**, no text can be entered into the field from the keyboard. If **lockText** is **false**, when you move the Browse tool over the field, the pointer changes to the I-beam and you can enter text. By default, **lockText** is set to **false**.

LockText can be useful when you want the information in fields to be protected so that the user cannot change that text. Also recall that if you use a **mouseUp** handler on a field, **lockText** must be set to **true**. If **lockText** is **false**, the click of the mouse button is used to place the vertical bar in the field so that text may be entered. In this case, your handler is never called.

There are three other properties of fields that have the values **true** or **false**: **showLines**, **wideMargins**, and **autoTab**. The value of **showLines** determines whether horizontal lines are visible in the field. When you create a new field, **showLines** is set to **false**. Suppose you have a button called "Enter text" on your card. You can write a handler that causes the lines in a field to appear so that the user can more easily enter text, for example,

```
on mouseUp
   set the lockText of card field "Text" to false
   set the showLines of card field "Text" to true
   beep
end mouseUp
```

WideMargins is used to leave extra space at the right and left edge of a field. In the following example, the text in both fields is left aligned. In the field on the left, **wideMargins** is set to **false**; in the field on the right, **wideMargins** is set to **true**. The extra space at the right edge of the field is harder to see when text is left aligned.


This field has wideMargins set to false so the text is close to the edge of the field.
This field has wideMargins set to true so the text is not as close to the edge of the field.

When you create a new field, **wideMargins** is set to **false.** You can add **wideMargins** to the previous handler:

```
on mouseUp
   set the lockText of card field "Text" to false
   set the showLines of card field "Text" to true
   set the wideMargins of card field "Text" to true
   beep
end mouseUp
```

Clicking on your button now provides a writing environment that looks much like a piece of paper.

When a new field is created, **autoTab** is automatically set to **false.** If **autoTab** is set to **true,** pressing Return on the last line of a nonscrolling field causes the text insertion point to move to the next field on the card. This feature can be useful if you want the user of your stack to be able to easily enter information into several fields. Instead of having to move the mouse and click on the "next" field, the user can simply press Return to move to the next field.

Technical Note: When ***autoTab*** *is* ***true****, the insertion point moves from the current card field to the next higher numbered card field. If there are no higher numbered card fields, it moves to the lowest numbered background field. Similarly, if the insertion point moves from a background field, it goes to the next higher numbered background field. If there are no higher numbered background fields, it moves to the lowest numbered card field.*

Like other field properties, **lockText, showLines,** and **wideMargins** can be determined for any particular field by typing statements such as

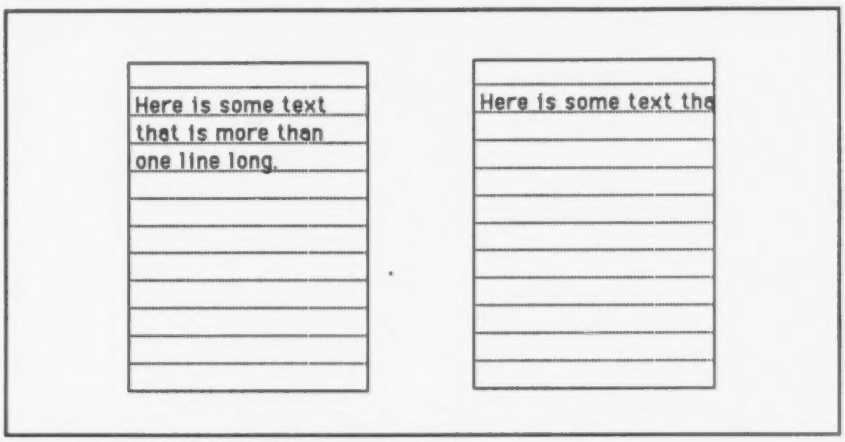
```
the lockText of card field "Demo"
the showLines of card field "Demo"
the wideMargins of card field "Demo"
```

into the Message box.

If you are using *HyperCard* 2.0, you also have the field properties **fixedLineHeight, dontWrap,** and **dontSearch.** When you create a new field, **fixedLineHeight** is set to **true,** while **dontWrap** and **dontSearch** are set to **false.**

FixedLineHeight works in tandem with **showLines. FixedLineHeight** must be **true** in order to make **showLines** be **true. DontSearch** is used to prevent a field from being

examined when you use **Find** to look for text in a stack. If **dontWrap** is **true**, a field in which the words wrap to the next line looks like the field on the left in the following illustration; but if **dontWrap** is set to **false**, the field looks like the one on the right:

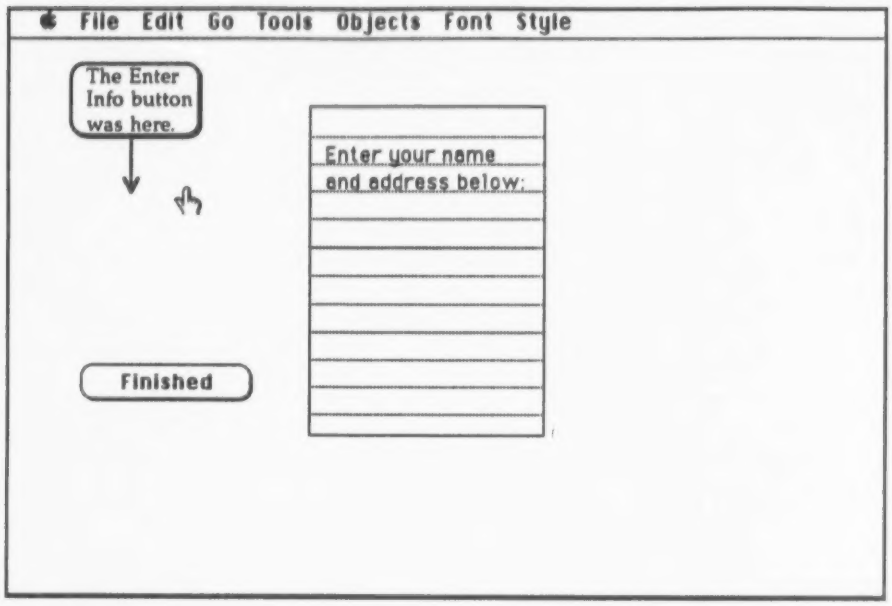


Working On Field Properties

You might make use of some of these field properties to create a card onto which only the first user can enter information. Start with a card with two buttons and one field. Initially only the Enter Info button is visible. The Finished button is to be hidden, as is a card field called Data. The script on the Enter Info button might read

```
on mouseUp
   show card field "Data"
   set the lockText of card field "Data" to false
   set the wideMargins of card field "Data" to true
   set the showLines of card field "Data" to true
   show card button "Finished"
   hide card button "Enter Info"
end mouseUp
```

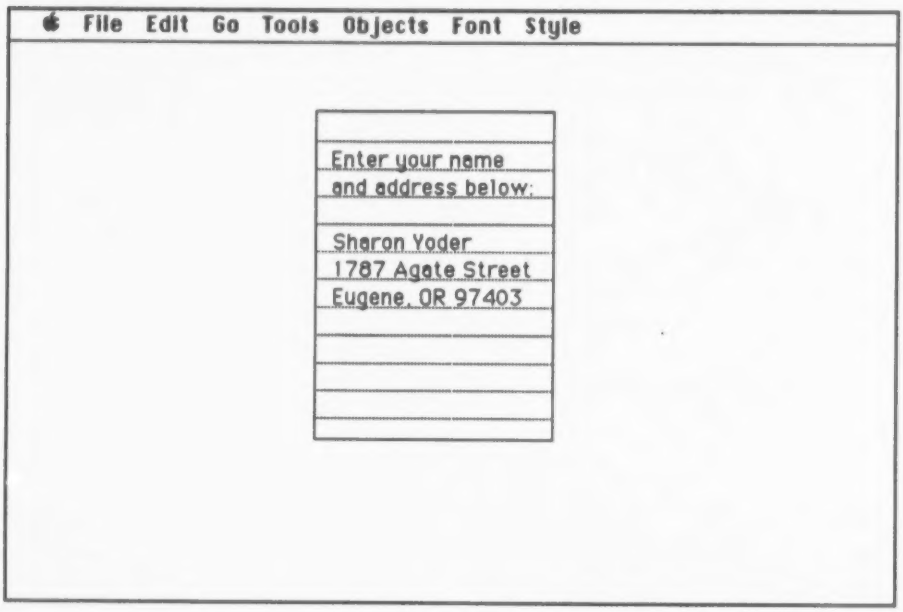
Clicking on Enter Info causes the Finished button to appear and the Enter Info button to disappear. It also causes the field, which contains a message, to appear:



The Finished button could then have the script

```
on mouseUp
   set the lockText of card field "Data" to true
   set the showLines of card field "Data" to false
   hide card button "Finished"
end mouseUp
```

The user of this stack would have "permanently" entered his or her name and address:



This idea could be used as a title card for a stack so that the author can easily enter some initial information that future users cannot easily alter.

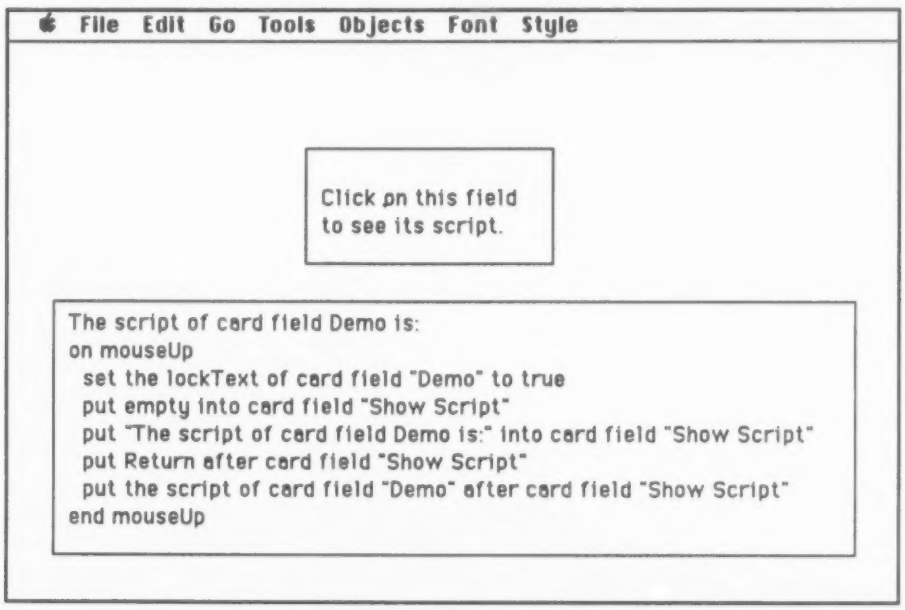
More With Field Properties

The **Font** button in the **Field Info** dialog box allows you to choose the style of type that will be used in your field. It is also possible to do this using HyperTalk commands. Working with fonts in fields will be discussed in Chapter 10.

Like the script for button, the script of a field can be accessed. You could use this idea to copy a script from one field to another. Suppose you have a card with two fields: Demo and Show Script. Suppose the script of Demo is

```
on mouseUp
   set the lockText of card field "Demo" to true
   put empty into card field "Show Script"
   put "The script of card field Demo is:" into card field¬
      "Show Script"
   put Return after card field "Show Script"
   put the script of card field "Demo" after card field¬
      "Show Script"
end mouseUp
```

Then, clicking on the field Demo causes its script to appear in the second field:


File Edit Go Tools Objects Font Style
Click on this field
to see its script.
The script of card field Demo is:
on mouseUp
set the lockText of card field "Demo" to true
put empty into card field "Show Script"
put "The script of card field Demo is:" into card field "Show Script"
put Return after card field "Show Script"
put the script of card field "Demo" after card field "Show Script"
end mouseUp

A more practical use of the **script** property might be to copy a script from one field to another. You might type

```
set the script of card field "Show Script" to the script of
   card field "Demo"
```

in the Message box. Both fields now contain the same script.

You were able to use **set** to determine the style of a button. Similarly, you can set the **style** property of fields. The **style** property of a field can have five different values:

```
transparent
opaque
rectangle
shadow
scrolling
```

For example, typing

```
the style of card field "Demo"
```

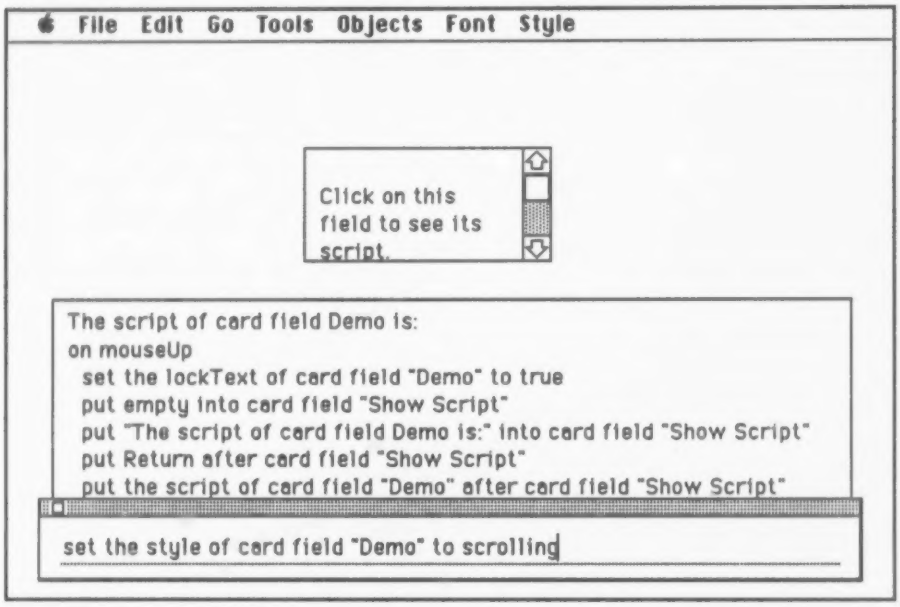
results in

```
rectangle
```

Of course, you can set the style of a field by typing such commands as

```
set the style of card field "Demo" to scrolling
```

In this case, you see


File Edit Go Tools Objects Font Style
Click on this field to see its script.
The script of card field Demo is:
on mouseUp
set the lockText of card field "Demo" to true
put empty into card field "Show Script"
put "The script of card field Demo is:" into card field "Show Script"
put Return after card field "Show Script"
put the script of card field "Demo" after card field "Show Script"
set the style of card field "Demo" to scrolling

A scroll bar has appeared on the right edge of the top field, Demo.

You can use both the **location** and the **rectangle** properties with fields. For example, you might type

```
the location of card field "Demo"
```

in the Message box, and then see something like

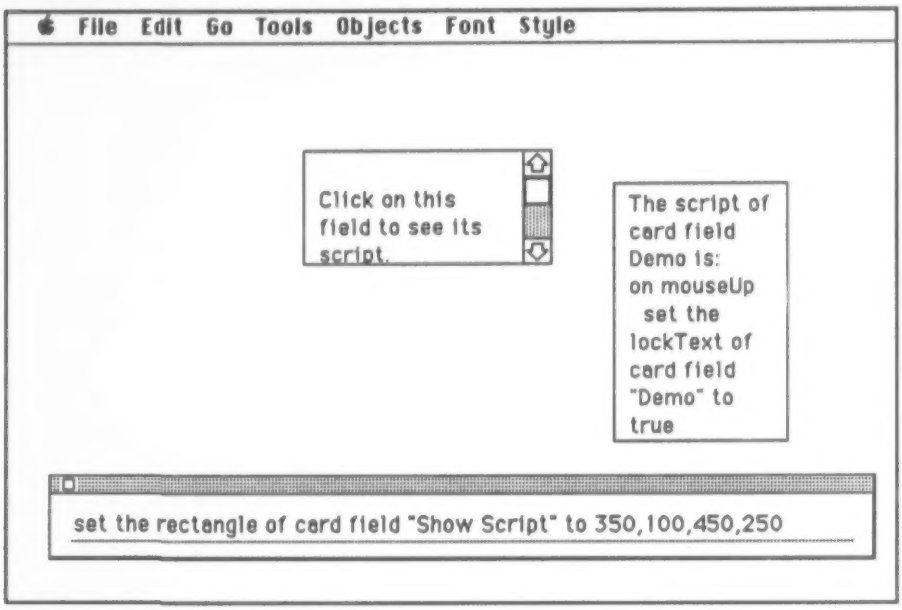
```
243,114
```

Recall that these numbers give the coordinates of the center of the field. You can also think of the first number as the distance from the left edge of the card to the center of the field and the second number as the distance from the top of the card to the center of the field.

Like **location**, **rectangle** can be used either to determine the current placement of a field on the card or to place a field. For example, you could type

```
set the rectangle of card field "Show Script" to
   250,150,400,300
```

and you would see that the field Show Script both changed in size and moved to a new location on the card:


File Edit Go Tools Objects Font Style
Click on this field to see its script.
The script of card field Demo is:
on mouseUp
set the lockText of card field "Demo" to true
set the rectangle of card field "Show Script" to 350,100,450,250

Debugging and Testing

Note that **showLines** does not work with scrolling fields. Lines will not appear in a scrolling field whether you set **showLines** to **true** or **false**.

Be careful when using the **scrolling** value for the **style** of a field. There is also a **scroll** *property* that is used to describe the number of pixels (points on the screen) that are scrolled off the top of the field.

When working with fields that have **mouseUp** or **mouseDown** handlers, be sure that the fields are locked. If they are not locked, the handlers do not work correctly. The **mouseUp** and **mouseDown** messages are "used up" by the placement of the insertion point.

When working with fields, keep in mind that a *line* in a field ends with an invisible carriage return. As you change the size of a field, word wrap causes different numbers of words to appear on each physical line of the field. However, the carriage return will always be the end of a block of text *and* will indicate where the *HyperCard* line ends.

New HyperTalk Words in This Chapter

Properties	
autoTab	rectangle
font	script
ID	scroll
location	showLines
lockText	style
name	wideMargins
number	
(*HyperCard* 2.0 only)	
dontSearch	fixedLineHeight
dontWrap	

Field Styles
opaque
rectangle
scrolling
shadow
transparent

Project Ideas

Create a card that has buttons to create different kinds of fields. Each button should have a script specifying a number of field properties. You should create buttons for the types of fields you like to use frequently. You can create fields on this utility card and copy them to cards you are working on.

Create a card that has students enter information such as name, favorite color, favorite pet, address, and so forth. You can follow the model given in this chapter in which buttons make the appropriate fields appear and then the buttons disappear once they are used. How might you use a stack with a card of information for each student in the class? Perhaps you could create a "yearbook" for your class. Or you could have each student create a grade card with a secret name that only you know. Then you could enter the grades using a button that puts the grade in the field and then locks the field.

Chapter 7. Overview of HyperTalk Grammar

This chapter focuses on the structure of the HyperTalk programming language. In particular, you will review what you have learned about commands, functions, and properties. In addition, you will begin to explore the concept of message passing.

Programming languages, like "natural" languages (English, French, Chinese, Russian, etc.), have rules of grammar. Natural languages can be very complex, taking many years for a nonnative speaker to learn. Programming languages are much simpler to learn than their natural-language counterparts. However, even among programming languages there is great variation. For example, the simplest versions of the BASIC programming language have only a couple of dozen special words to learn, while modern versions of the Logo programming language often have more than 200 special words. On the other hand, the syntax rules of the Logo programming language can be described in a few paragraphs, while the rules for Pascal can take pages of diagrams. In addition, some languages are virtually without special cases. That is, each and every statement in the language follows the same simple set of rules of the language exactly. Natural languages, on the other hand, tend to be filled with exceptions and special cases. Similarly, some programming languages have a number of special cases that often exist to make working with the language easier for the computer. However, the programmer must remember these special cases in order to write correct programs.

The HyperTalk programming language is neither the simplest nor the most complex among programming languages. There are some "special cases" in HyperTalk, but in general it follows its own grammatical rules quite well. While it has a large number of special words, they are very "English like" and thus relatively easy to remember.

Commands and Functions

Earlier in this book, you learned about both commands and functions in HyperTalk. Recall that commands perform an action immediately, while functions merely generate values that are either put in containers like the Message box or used in HyperTalk statements.

Consider the **play** command. The following HyperTalk statement causes three notes to be played:

```
play "boing" "d e f"
```

If you type this statement in the Message box, the notes are played immediately. **Play** is a command that causes an immediate action.

If you type

```
the number of card buttons
```

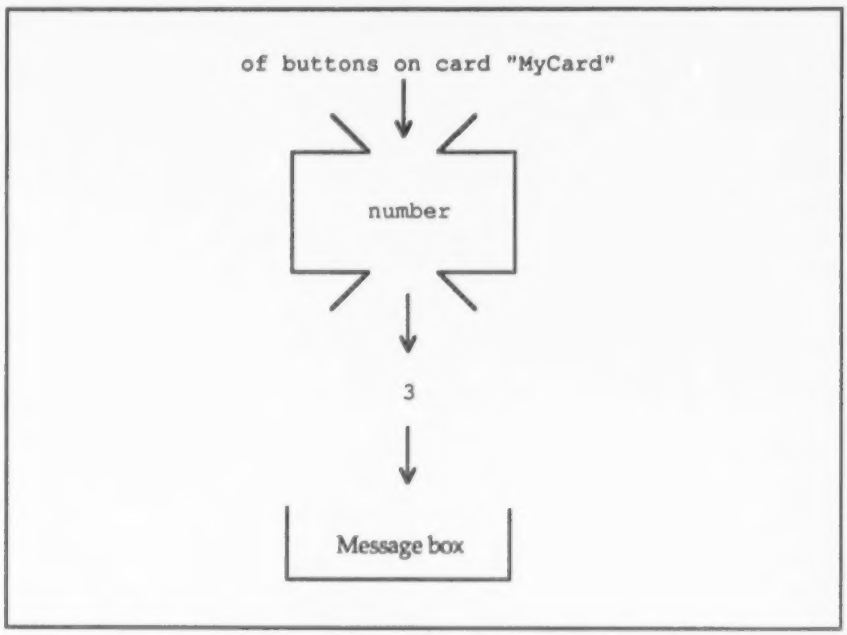
the **number** function produces a value representing the number of buttons on the card. When you type this function in the Message box with no command preceding it, *HyperCard* assumes the result should be displayed in the box.

Of course you can use a command to tell HyperTalk what to do with the value produced by the function:

```
put the number of card buttons into card field 1
```

In this case, the function **number** produces a value, and that value is "given to" the command **put**, which then inserts that number into the appropriate card field.

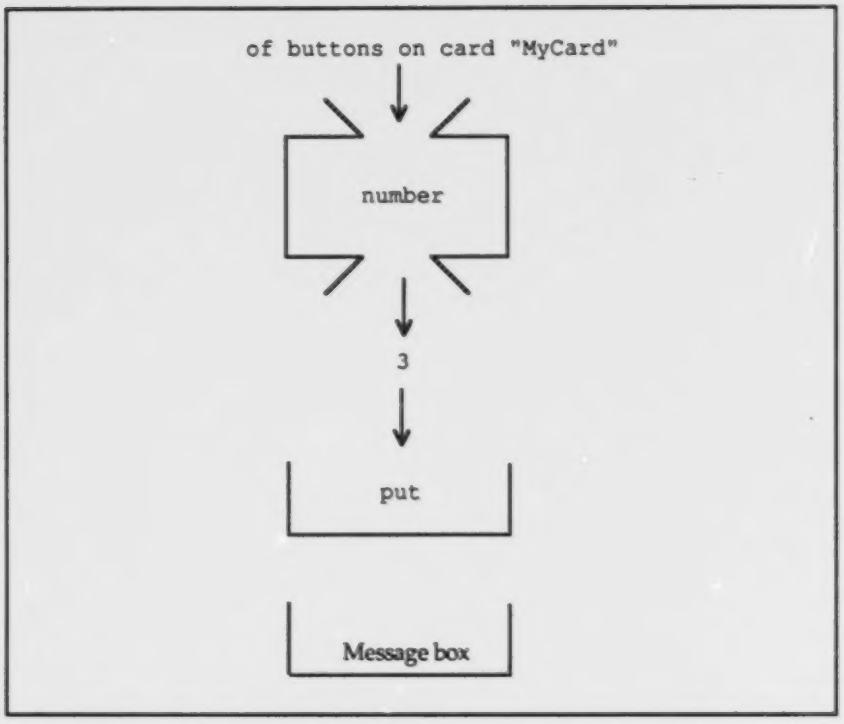
You might think of functions as "funnels." If they have parameters, they accept values from their parameters and produce a result. If there is no command to "catch" the result, that result automatically is sent to the Message box:



However, if the **put** command is present, for example in

```
put the number of card buttons in card field 1
```

the result of the function is used by **put** to display the value in the card field:



The Message box is not needed to "catch" the value from the function. Take a few moments to think about the difference between commands and functions.

Syntax Descriptions

In this book, syntax descriptions are given for each new command and function that you learn. Take a few minutes to review these syntax descriptions. Here is the complete description of the **go** command, which you saw in Chapter 1. Recall that commands are shown in a single-line box.

```
go [to] [stack] stackName
go [to] bkgndDescriptor [of [stack] stackName]
go [to] cardDescriptor [of bkgndDescriptor] [of [stack]
   stackName]
```

In each of the preceding versions of **go**, you see that the word "to" is optional since it is in square brackets. Note that if you are referring to a stack only, the word "stack" is optional, but *stackName*, the legal name of a stack, must be included. On the other hand, *cardDescriptor* is the word "card" followed by a legal descriptor of a card, while *bkgndDescriptor* is the word "background" followed by a legal descriptor of the background.

To summarize, in syntax diagrams in this book:

- Anything in square brackets–[ ]–does not have to be included, e.g., "stack" in the previous diagram.
- Words in italics are called *parameters*, and particular values must be substituted for these parameters, e.g., stackName is a parameter.
- There can be optional words inside optional phrases, e.g., in the last form of the previous **go** command, the word "stack" is optional within the optional phrase "of [stack] *stackName*."

Take a few minutes to examine the **go** command in more detail. First, look at the first form of the **go** command:

```
go [to] [stack] stackName
```

There is a variety of ways to use this form:

```
go "My Stack"
go stack "My Stack"
go to "My Stack"
go to stack "My Stack"
```

In the second form of the **go** command

```
go [to] bkgndDescriptor [of [stack] stackName]
```

there is an optional phrase

```
[of [stack] stackName]
```

That is, if you wish to use **go** to move to a background, there are two possible optional phrases (underlined in the following commands) that you can insert after the background descriptor:

```
go to background "Frame" of "My Stack"
go to background "Frame" of stack "My Stack"
```

Note that once you choose to use the "of" phrase, you can either use or omit the word "stack" but you must include the specific stack name.

The third form of the **go** command allows you to go to a card:

```
go [to] cardDescriptor [of bkgndDescriptor] [of [stack]
   stackName]
```

A command to go to a card can take a variety of forms, some of which include optional phrases (underlined in the following commands), for example,

```
go to card "Picture"
go to card "Picture" of "My Stack"
go to card "Picture" of background "Frame"
go to card "Picture" of background "Frame" of "My Stack"
```

It is important that you learn to read syntax descriptions such as these so that you are able to understand the correct form of HyperTalk commands and functions and so that you can debug your HyperTalk programs more easily.

Next, examine a syntax description for a function. Recall that functions are shown in a box with double lines. The syntax description for the **number** function looks like this:

```
[the] number of objects
[the] number of chunkType in string
```

Here, *objects* can be

```
[background] buttons
[card] fields
backgrounds
cards
```

while *chunkType* can be

```
characters (or chars)
words
items
lines
```

None of the parts of the **number** function is optional. The first form,

```
[the] number of objects
```

is used to count *HyperCard* objects, such as cards, buttons, or fields, for example,

```
the number of background buttons
the number of cards
```

Note that there is a specific list that must be substituted for *objects* in the syntax description. In this book, such lists will always be given just below the box.

The second form of **number**,

```
[the] number of chunkType in string
```

is used to count HyperTalk characters, words, items, and lines. *String* is used to tell HyperTalk where to look. For example,

```
the number of lines in card field "Data"
```

puts a number representing the number of lines that are found in the card field named Data.

It seems as if every HyperTalk command has some different words representing the different optional and required parts. These words represent *values*, and the words used are just to help you understand what *kind* of value goes in each parameter. Some of these words are merely descriptive, like "cardDescriptor" in the **go** command above. Others, like "chunk," have a technical meaning in HyperTalk. You will learn more about some of these technical meanings later.

Using English-like words in syntax descriptions makes them fairly easy to read and understand. However, computer languages are very precise so that "literal-minded" computers can understand them. On the other hand, natural languages like English do not require a high degree of precision to be understood. When a computer language is described in an English-like manner, it is hard to make the precise meaning of the computer language clear. There are a number of techniques that computer scientists use

to specify computer languages exactly. One such technique is called Bakus-Naur notation. The Bakus-Naur notation for the **go** command looks like this:

```
go [to] [[<ordinal> | <position>] <endLine> | <dest>] [in [a]
   new window] [without dialog]
```

where each word or phrase in angle brackets (< >) is specified by another Bakus-Naur expression. A person who understands this notation can easily determine exactly what the "legal" forms of **go** are. However, this notation is not very readable for a novice. Thus, this book sacrifices precision for readability in its syntax descriptions.

Objects

Almost all programming languages have commands and functions. The tasks that you can accomplish using the language revolve around creating instructions using the commands and functions available in the language. However, *HyperCard* is somewhat different from languages like Pascal and BASIC. *HyperCard* is related to a category of programming languages called "object-oriented" languages. These languages rely on the concept of an *object*, as does *HyperCard*. In a true object-oriented programming environment, programmers can create their own objects. However, the *HyperCard* programmer is limited to a variety of forms of the five objects provided by *HyperCard*: buttons, fields, cards, backgrounds, and stacks. Thus, *HyperCard* is often referred to as an "object-like" programming environment.

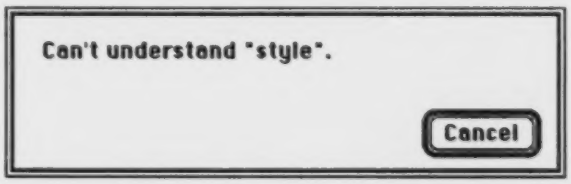
Properties

You have learned that in *HyperCard* the characteristics that "describe" buttons and fields are called *properties*. Each object in *HyperCard* has a number of different properties describing that object. If you make a copy of an object, the new object "inherits" the properties of the original. That is, if you make a rectangular button with the name shown, any copy of that button will also be rectangular and will have the same name.

Properties are used to get or change information about a *HyperCard* object. You can't use properties by themselves. For example, if you type

```
style
```

in the Message box, you see



Instead, property words are used to get information about an object, for example,

```
the style of card button "Experiment"
```

or to change some property of an object, for example,

```
set the style of card button "Experiment" to "transparent"
```

Values

The HyperTalk values used in commands and functions come from a number of sources: constants, literals, numbers, functions, properties, and containers. You have seen the constant **empty**. Remember that **empty** represents a text string with nothing in it. You saw statements such as

```
put empty into card field "Data"
```

Values also come from literals. You have seen literal text strings like "Hi" used earlier in the book, for example,

```
put "Hi" into the Message box
```

Numbers are also values, for example,

```
put 5 into the Message box
```

In addition, values come from functions and properties. **Number** reports a value, which is then put in the Message box, for example,

```
the number of card fields
```

or used by a command, such as

```
put the number of card fields into the Message box
```

Similarly, values come from examining the properties of a *HyperCard* object. So, if you type

```
put the location of card button "Experiment" into the¬
   Message box
```

the value of the **location** is put into the appropriate card field.

Values can also come from containers. You have learned that the Message box is a one-line container. You have seen that text can be put in fields and that fields can act as containers. You can move a value from one container to another, for example,

```
put line 1 of card field "Data" into the Message box
```

You will learn about other kinds of containers in Chapter 23.

Messages

Object-oriented programming languages have the characteristic that "messages" can be sent from one object to another. Message passing is an important idea in *HyperCard*. To completely understand how *HyperCard* works, you must understand how message passing works.

You have already worked with message passing, but you are probably unaware of what is happening. When you press the mouse button down, a **mouseDown** message is sent to the object directly underneath the pointer. When you release the mouse button, a **mouseUp** message is sent. If you have a button with the script

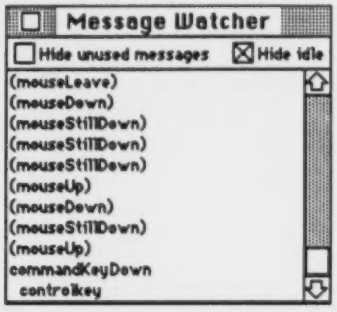
```
on mouseUp
   beep 5
end mouseUp
```

you know that clicking on the button causes five beep sounds. Recall that the previous code (beginning with **on** and ending with a line that starts with **end**) is called a *handler*. Handler is an abbreviation for "message handler." A **mouseUp** handler looks for and then "handles"—makes use of—a **mouseUp** message.

If you are using *HyperCard* 2.0, you can observe the constant flow of messages as *HyperCard* is running. In the Message box, type

```
show the Message Watcher
```

The following window will appear:



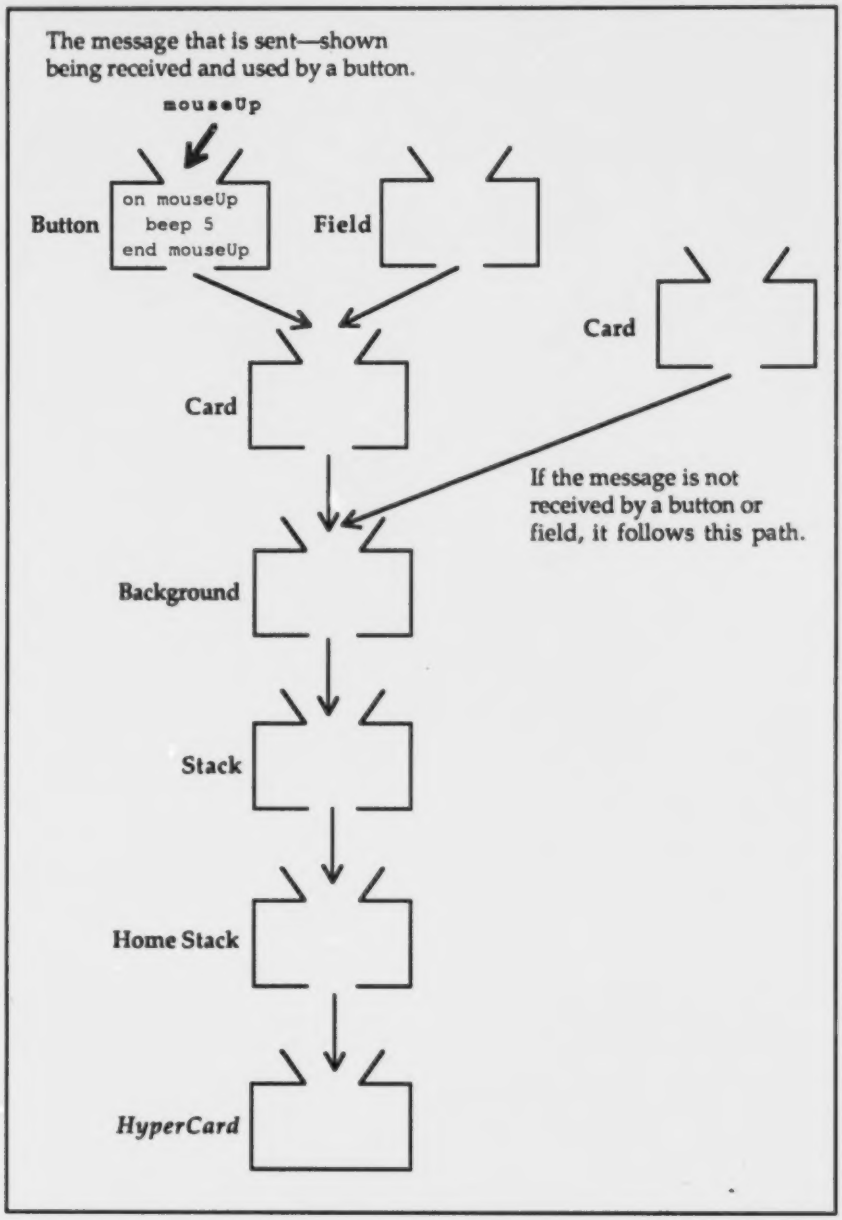
As you move the mouse about, press keys, or press the mouse button, messages appear in the Message Watcher window. For more details about this window and other debugging features of *HyperCard* 2.0, see your documentation.

You have only worked with very simple cases making use of *HyperCard* messages. Recall, however, that in Chapter 3 you were cautioned to lock a field containing a **mouseUp** handler. If the field is not locked, the **mouseUp** message is "used" by the field to place the insertion point so that you can enter text in the field, and the handler on the field does not receive the message. This illustrates the importance of understanding the flow of messages within *HyperCard*.

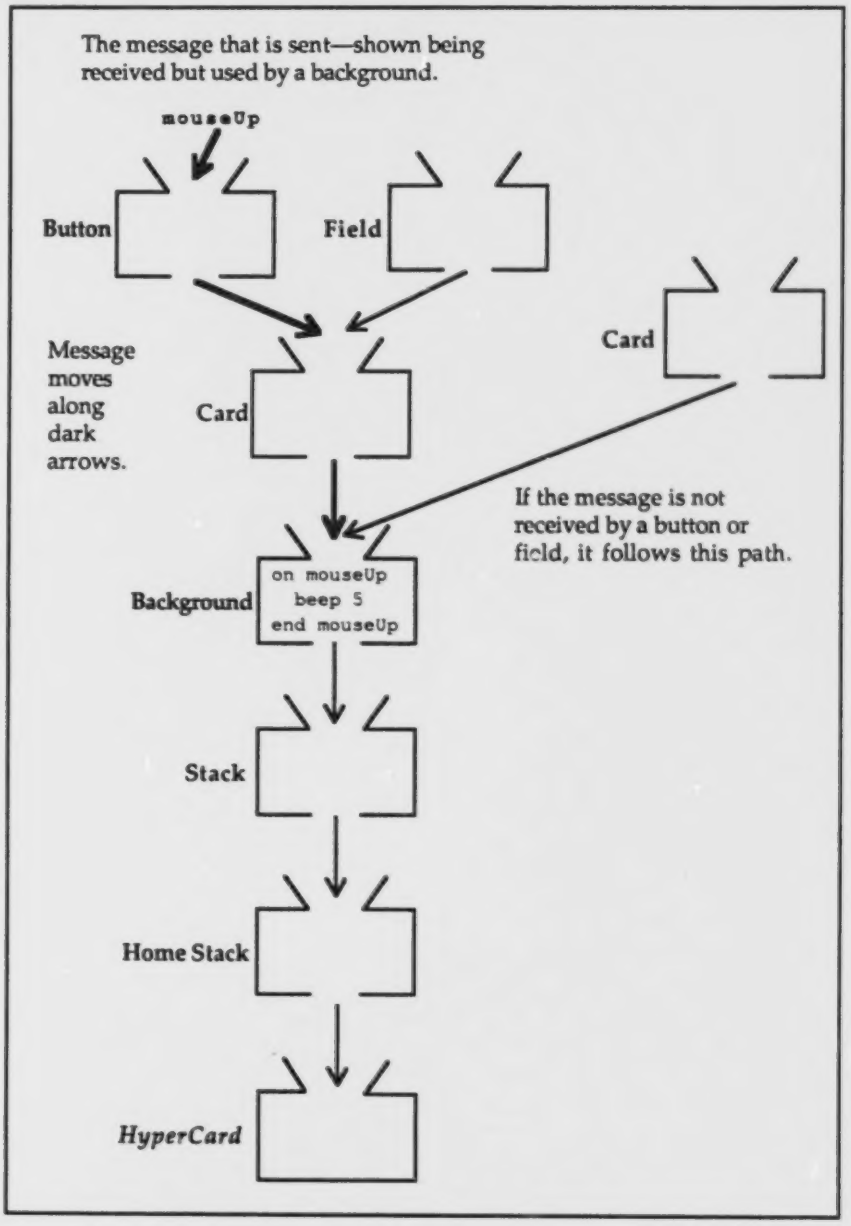
Messages are constantly and automatically moved from one object to the next within *HyperCard*. You are only aware of these messages when your handler makes use of them. A message flows along a specific and predefined path until either a handler is found or the message reaches *HyperCard* itself.

To begin to understand message passing, examine the following example. Suppose a **mouseUp** message has been sent by releasing the mouse button when the mouse pointer is over a button. The following diagram illustrates the path that messages can take in *HyperCard*. A message may be received by a button, field, or card. The card box on the right represents the case in which the mouse is clicked over the card itself rather than over a button or field. In the case of the mouse button being clicked while the pointer is

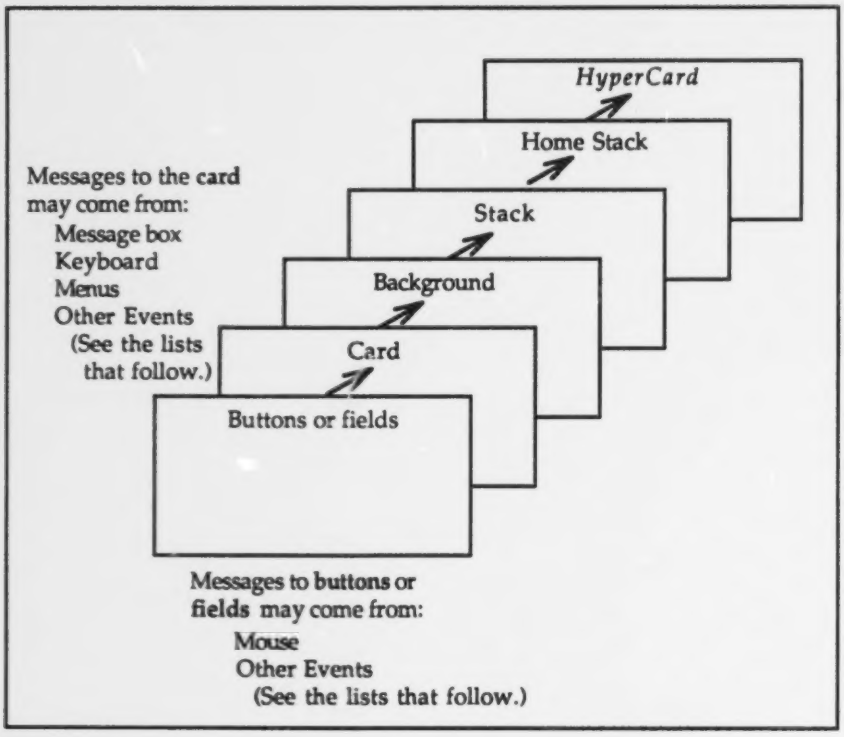
over a button, the **mouseUp** message is received by the button. Since there *is* a handler for **mouseUp**, the message is used there and does not "fall through" to another object.



Note that if there is no **mouseUp** handler on the button but there is a **mouseUp** handler on the background, the **mouseUp** message "falls through" until it reaches the background:



The following illustration is a different kind of diagram that shows the message-passing hierarchy. Spend some time examining this diagram. Note that messages are sent from a number of places: the keyboard, the Message box, menus, and the mouse.



Messages come from a variety of sources. Messages that send messages to the *card* include:

`arrowKey`	`functionKey`	`openBackground`
`closeBackground`	`help`	`openCard`
`closeCard`	`hide`	`openStack`
`closeStack`	`idle`	`quit`
`controlKey`	`mouseDown`	`resume`
`deleteBackground`	`mouseStillDown`	`returnKey`
`deleteCard`	`mouseUp`	`show`
`deleteStack`	`newBackground`	`startUp`
`doMenu`	`newCard`	`suspend`
`enterKey`	`newStack`	`tabKey`

Note that the mouse messages are sent to the card only when there is no button or field to receive them.

Messages sent to buttons include

deleteButton	mouseLeave	mouseWithin
mouseDown	mouseStillDown	newButton
mouseEnter	mouseUp	

Messages sent to fields include

closeField	mouseLeave	newField
deleteField	mouseStillDown	openField
mouseDown	mouseUp	tabKey
mouseEnter	mouseWithin	

It is beyond the scope of this book to discuss the details of all of the conditions under which each of these messages is sent to a particular *HyperCard* object. However, you will learn about a number of these messages in subsequent chapters.

Technical Note: If you wish to pursue your study of HyperTalk, you will need a reference such as Apple *HyperCard Script Language Guide—The HyperTalk Language published by Addison-Wesley, or Winkler and Kammins'* HyperTalk 2.0 —The Book, *published by Bantam Books.*

Scripts

Commands and functions, properties and values: these are all used in the construction of HyperTalk statements that are put together into small programs. In HyperTalk, programming is usually called *scripting*. Each *script* is associated with a particular *HyperCard* object. These scripts are made up of one or more *handlers*. It is the handlers that receive and act on the messages that are automatically sent from object to object in *HyperCard*. A handler always begins with the word **"on"** and ends with a line beginning with **"end."** These words are called *keywords*. You will learn about other keywords later in this book.

The handlers that make up the script of an object do not have to be in any particular order in the script editor. However, if you have two handlers that use the same message, HyperTalk will run the first one and ignore the second. That is, suppose you have these handlers on the same button:

```
on mouseUp
   beep
end mouseUp

on mouseUp
   beep 3
end mouseUp
```

If you click on the button, you hear one beep. However, if you have the handlers arranged like this

```
on mouseUp
   beep 3
end mouseUp

on mouseUp
   beep
end mouseUp
```

and click on the button, you will hear three beeps.

Debugging and Testing

When writing and using handlers, you must think carefully about where you put them. Suppose, for example, that you have the handler

```
on mouseUp
   beep 2
end mouseUp
```

on a button. On the card you have the handler

```
on mouseUp
   beep 3
end mouseUp
```

If you click on the button, you hear two beeps. If you click anywhere else on the card, you hear three beeps. Do you understand why? Do you see that the **mouseUp** message is "used" by the button if the mouse pointer is over the button, so there is no message for the card handler to act on?

Occasionally you have handlers that don't seem to work. Most likely they are not receiving a message to act on, just as the card in the previous example would not get a **mouseUp** message when you clicked over the button. Problems like this can occur if you have one object on top of another. Perhaps the top object is using the message, so it is never being received by the correct object. Note also that buttons and fields do not pass messages along to each other. That is, if you have a card button on top of a background field, a message not used by the button will not be passed to the field. Rather, it will move from the button to the card.

You have learned the command **put**. When used with the Message box, **put** can be a valuable aid to debugging. You can check property values. For example,

```
put the icon of card button 1 into the Message box
```

lets you easily determine the icon number of a particular button. You can use functions to generate values to display in the Message box. For example,

```
put the number of card buttons into the Message box
```

tells you how many buttons there are on a card, including invisible buttons. Later you will see that using **put** can help you determine the current values in containers.

While some of the preceding commands do not need **put**, the idea of checking the status of objects, the values of functions, or the values of containers using the Message box becomes increasingly more powerful as you progress in your programming in HyperTalk.

Have you noticed that the name of this section in each chapter is "Debugging and Testing"? Technically there is a distinction between these two ideas. Debugging means that you know there is a problem and you are trying to figure how what the problem is and how to fix it. Testing is checking to see if the code works correctly. It is the process of looking for bugs. When you check an individual handler or button to see if it works correctly, you are doing "testing." When your handler doesn't work correctly, you use debugging strategies to locate the problem and fix it.

New HyperTalk Words in This Chapter

System Messages (to Cards)		
arrowKey	functionKey	openBackground
closeBackground	help	openCard
closeCard	hide	openStack
closeStack	idle	quit
controlKey	mouseDown	resume
deleteBackground	mouseStillDown	returnKey
deleteCard	mouseUp	show
deleteStack	newBackground	startUp
doMenu	newCard	suspend
enterKey	newStack	tabKey

System Messages (to Buttons)		
deleteButton	mouseLeave	mouseWithin
mouseDown	mouseStillDown	newButton
mouseEnter	mouseUp	

System Messages (to Fields)		
closeField	mouseLeave	newField
deleteField	mouseStillDown	openField
mouseDown	mouseUp	tabKey
mouseEnter	mouseWitnin	

Project Ideas

Spend some time thinking about the message-passing diagram given in this chapter. For example, what happens if a button has this handler,

```
on mouseUp
   beep 5
end mouseUp
```

and the mouse is clicked over the button? What if the mouse is clicked outside of the button? Move the handler to a field. What happens if a button is on top of the field? Look at other scripts in this book. Can you trace what happens to the messages created by the keyboard, in the Message box, from menus, or from the mouse?

To explore message passing, put scripts on various objects that tell you where the message was received. For example, you might put

```
on mouseUp
   put "on background button Demo1" into the Message box
end mouseUp
```

on the background button called Demo1. After you have put scripts like this on a number of objects, experiment with clicking at different points on your card.

Chapter 8. More About Properties

In this chapter, you will learn about properties that are not associated with a specific *HyperCard* object. In particular, you will learn about global properties and window properties.

In Chapter 4, you learned about the properties associated with buttons. In Chapter 6, you learned about the properties associated with fields. You learned how to find the values of these properties and how to change some of them with the **set** command. In subsequent chapters you will learn about the properties associated with the other *HyperCard* objects: cards, backgrounds, and fields. In addition to object properties, there are a number of other kinds of properties that are useful when you are programming in HyperTalk.

Global Properties

There are some properties that can be used anywhere in your stack. That is, they remain the same as you move from card to card unless you change them. You are familiar with a number of global properties because they can be set from the Home stack. In order to write scripts, you must have the user level set to 5. You most likely did this by using the Preferences card in the Home stack. **UserLevel** is a global property. You can check or change the **userLevel** from places other than in the Home stack. Make sure you are *not* on the Preferences card in the Home stack. Type

```
the userLevel
```

in the Message box. You should see the number 5. Recall that you must precede the name of a property by the word "the" to get its value in the Message box.

If you now type

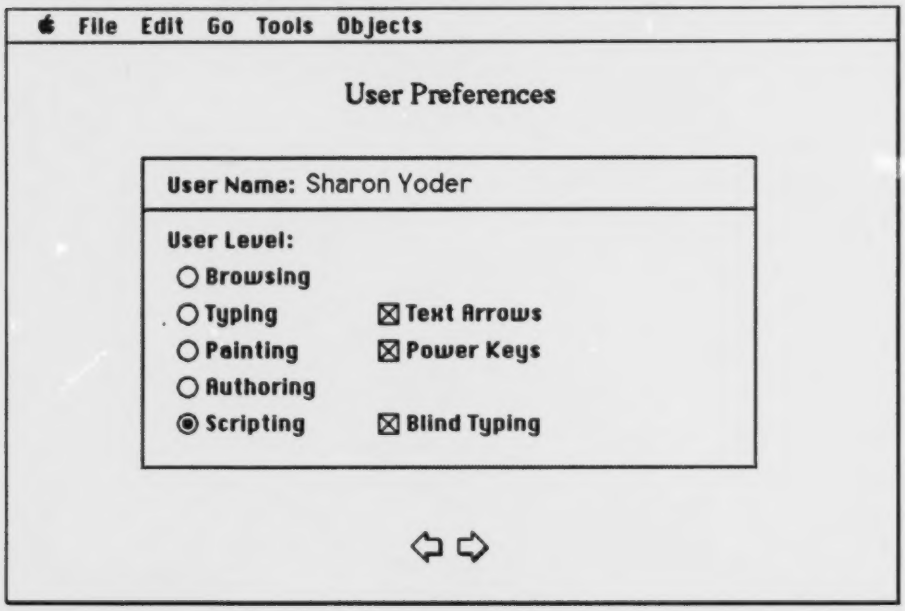
```
set the userLevel to 1
```

and then try to type in a field, you will see that you can no longer change the contents of a field. Level 1 is "browsing," while level 5 is "scripting." Go to a different card and type

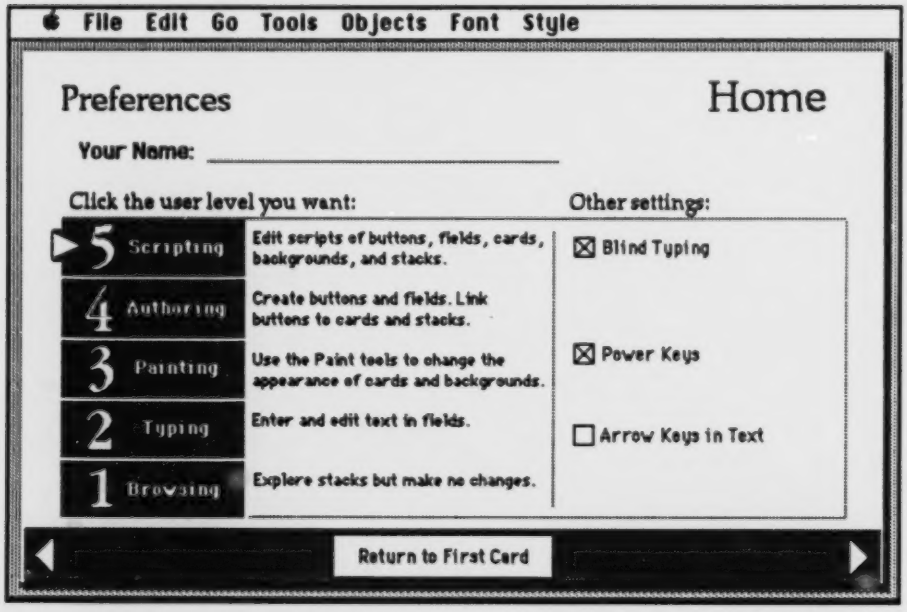
```
the userLevel
```

in the Message box. You see that the value is 1 no matter what card you are on. **UserLevel** is *global* and remains the same throughout your session with *HyperCard*. (Be sure to set the **userLevel** back to 5 before you continue.)

There are three other global properties you can set on the Preferences card. In the versions of *HyperCard* before 2.0, this card looks something like this:



Note that Text Arrows did not exist before *HyperCard* version 1.2. In *HyperCard* 2.0, this card looks like this:


File Edit Go Tools Objects Font Style
Preferences
Home
Your Name:
Click the user level you want:
Other settings:
5 Scripting
Edit scripts of buttons, fields, cards, backgrounds, and stacks.
4 Authoring
Create buttons and fields. Link buttons to cards and stacks.
3 Painting
Use the Paint tools to change the appearance of cards and backgrounds.
2 Typing
Enter and edit text in fields.
1 Browsing
Explore stacks but make no changes.
Blind Typing
Power Keys
Arrow Keys in Text
Return to First Card

If you type

```
set blindTyping to true
```

you can type HyperTalk statements without the Message box being visible. This is equivalent to checking the "Blind Typing" box on the Preferences card. Typing

```
set powerKeys to true
```

is the equivalent of checking the Power Keys box on the Preferences card. When **powerKeys** is **true**, a number of keyboard shortcuts that are useful when working with graphics become available. (See the documentation or Help stack with your version of *HyperCard* to determine what power keys are available to you.)

The third global property available on the Preferences card is

```
textArrows
```

Setting **textArrows** to **true** is the equivalent of checking the "Text Arrows" or "Arrow Keys in Text" box on the Preferences card. If **textArrows** is **false**, the Right and Left Arrow keys take you to the next and previous cards in your stack. The Up and Down Arrows take you forward and back through the cards you have already looked at in your stack. With **textArrows** set to **false**, you can't use the Arrow keys when editing text in a field. If you set **textArrows** to **true**, you must hold down the Option key with the Arrow keys to move through your stack.

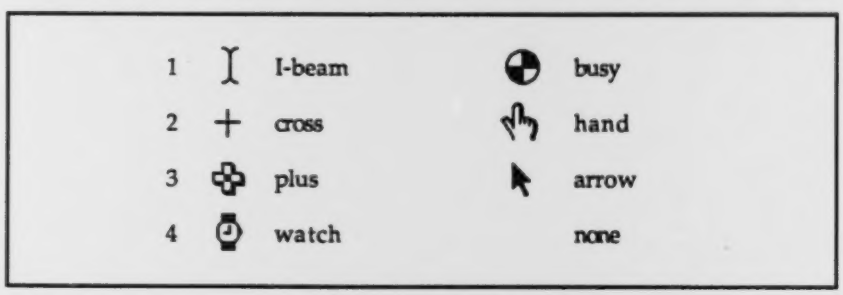
Another interesting global property is

```
cursor
```

This property determines the shape of the pointer. You have certainly noticed the ways in which this pointer changes as you move your mouse in *HyperCard* or in other Macintosh applications. Unlike the other properties you have learned about, you cannot determine the current **cursor** property by using the Message box. However, you can use **set** to change the **cursor** property. For example, you might script a button as follows:

```
on mouseUp
   set the cursor to crossbar
   wait for 3 seconds
end mouseUp
```

When you click on the button, you should see the pointer change to a crossbar. Standard pointers for *HyperCard* are I-beam, cross, thick plus, a watch indicating "wait," a circle indicating "busy," a hand, an arrow, and "none" for no cursor. The first four of these can be referred to by number. The pointer that appears when you use **set** with **cursor** and the appropriate predefined number or name is shown in the following box:



There are several global properties that allow you to control some of the normally automatic actions of *HyperCard*. For example,

```
lockRecent
```

controls what appears on the Recent card that you can access from the **Go** menu or by using Command-R. **LockRecent** is normally **false.** If you set it to **true,** miniature card

representations are not added to the Recent card. Setting **lockRecent** to **true** can make your scripts run faster.

The

```
lockMessages
```

property can also be used to make your scripts run faster. If you set **lockMessages** to **true**, automatic messages, such as **openCard, closeCard, newCard,** and **deleteCard**, are not sent. However, *HyperCard* automatically resets this property to **false** when current handlers stop running. Thus, this property is useful only when used within a handler.

The property

```
lockScreen
```

is used to prevent *HyperCard* from updating the screen when you go to another card. Thus, by setting **lockScreen** to **true** within a handler, you can go to another card—for example, to get some information—and return to the current card without the user seeing the data card. **LockScreen** is reset to **false** when the current handlers stop running. Setting **lockScreen** to **true** when you are going to several cards can also be used to make your scripts run faster.

Window Properties

When your are working with *HyperCard*, you use a variety of different windows. These windows have properties you can work with. The windows you have worked with include

```
card window
message (or msg) [box or window]
pattern window
tool window
```

In *HyperCard* 2.0, you have also seen the window

```
message watcher
```

You have seen the Message box and the Tool window. If you have worked with graphics very much in *HyperCard*, you have probably used the Pattern window. The term "Card window" simply refers to the *HyperCard* card you see on the screen.

Each of these windows has three properties you can modify:

```
visible
loc[ation]
rect[angle]
```

The **visible** property determines whether a particular window is shown or hidden. **Visible** can have the value of either **true** or **false.** Thus, you might type

```
set the visible of the tool window to true
```

This is equivalent to using the mouse to tear off the window from the Menu bar. Or, you might type

```
set the visible of the pattern window to false
```

This is equivalent to clicking on the close box of the Pattern window. It is also the equivalent of typing

```
hide the pattern window
```

When you use **show,** the **visible** of the window is set to **true,** and when you use **hide,** the **visible** of the window is set to **false.** It is not uncommon for there to be more than one way to produce the same result when working with HyperTalk. Similarly, you can

show or hide the Message box, the Card window, and the Pattern window by using **visible**.

Recall that you can specify the location of a button or a field using the **location** property. Similarly, you can indicate where a particular window will appear by using the **location** property. Thus,

```
set the location of tool window to 50,20
```

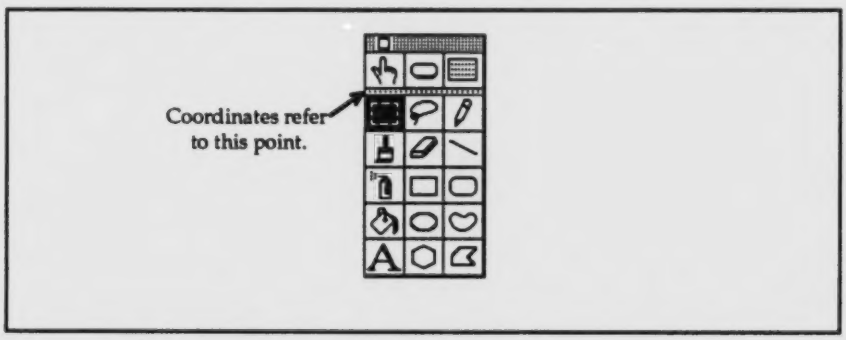
places the Tool window in the upper left side of the card, while

```
set the location of tool window to 400,200
```

places it in the lower right corner of the card. You can also use **location** to determine where a window is currently located. Just typing

```
the location of tool window
```

in the Message box causes two coordinates to appear. Recall that **location** gives you the center of a button or a field. For the Tool window, **location** gives the coordinates of the left end of the bottom of the bar across the top of the window.



The **location** of other windows, such as the Pattern window and Card window, are also just below the bar across the top.

The third property, **rectangle**, is preset for all of the *HyperCard* windows that you can manipulate from HyperTalk. The only exception is the Card window in *HyperCard* 2.0. In all other cases, you cannot **set** the **rectangle** property.

However, you can use this property to find out the part of the screen covered by a particular window. For example, if you type

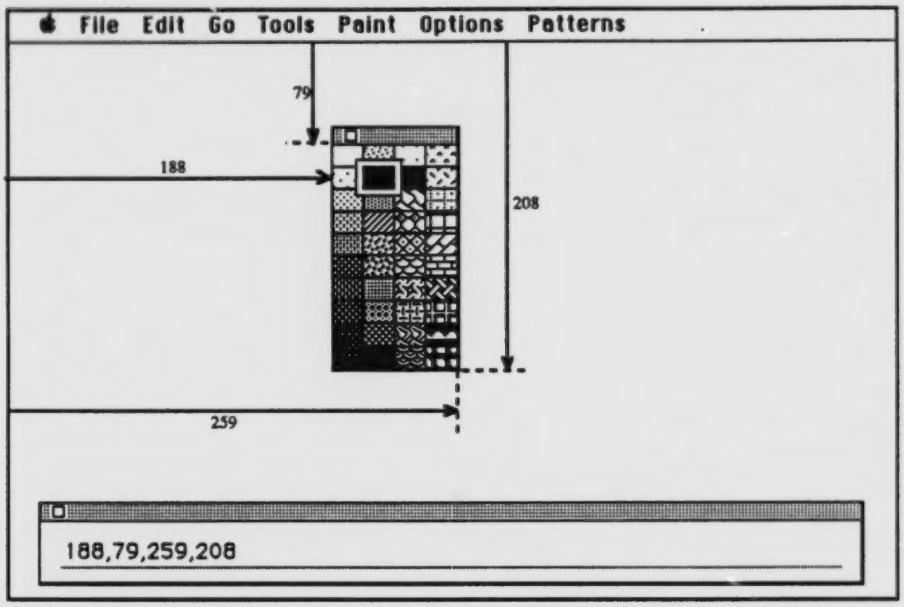
```
the rectangle of the tool window
```

in the Message box, you see four numbers. If you place the Tool window in the middle of the card, you might see

```
100,79,259,208
```


BEST COPY AVAILABLE

These four numbers represent the coordinates of the upper left and bottom right corners of the window, as shown in the following illustration:



That is to say, the left corner just above the patterns is 188 pixels from the left side of the card and 79 pixels from the top of the card. The lower right corner of the window is 259 pixels from the left edge of the card and 208 pixels from the top of the card.

Debugging and Testing

If you are using a larger Macintosh screen, you will find that you can move the cards to different locations on the screen. Your *HyperCard* stack can appear in the upper left corner of the screen or over at the right edge of the screen—wherever you want to place it. When working with coordinates, it is important that you understand that the origin—the point (0,0)—is always at the top left corner of the *card*. Thus, for example, suppose you place the Tool window on the card by typing

```
set the location of the tool window to 200,104
```

Then you move the entire card. If you again type

```
set the location of the tool window to 200,104
```

the Tool window stays at the same relative location on the card. In *HyperCard*, the coordinates of all of the windows except the Card window are "measured" from the upper left corner of the card. Remember also, that the top left corner of the card is just below the bar across the top of the card.

Note that the property **userLevel** is one word. A common error is to type **userLevel** as two words.

New HyperTalk Words in This Chapter

Properties (global)	
blindTyping	lockScreen
cursor	powerKeys
lockMessages	textArrows
lockRecent	userlevel

Properties (window)
loc[ation]
rect[angle]
visible

System Messages
closeCard
deleteCard
newCard
openCard

Cursors		
1	I-beam	busy
2	cross	hand
3	plus	arrow
4	watch	none

Windows
card
message [box or window]
pattern
tool
In *HyperCard 2.0:*
message watcher

Project Ideas

Try creating some buttons to place the Tool or Patterns windows or Message box at various places on the screen. You might create a "utility" card that lets you place these windows at different spots before you begin an exploration. For example you might have a "Left tools" button that puts the Tool window on the left side of the card and "Right tools" button that puts it on the right.

This might be a good time to create a Utilities stack. You could include cards suggested in earlier project ideas in this book. For example, you could have a card demonstrating visual effects. You could have a card that had examples of kinds of buttons you like to use so that you could easily cut and paste them into a new stack. You could include the suggestion from this chapter for creating a card that places various windows in locations where you like to have them. You might even include a few blank cards just for experimentation.

Spend some time determining exactly what the coordinates that are given by **location** and **rectangle** represent. Is the upper left corner of the card above or below the Menu bar? What about the Message box? (Remember, you can place the pointer at a location on the screen and then type **the mouseLoc** in the Message box to see the current coordinates of the mouse pointer.)

Experiment with changing some of the global properties. Which way do you prefer to have **blindTyping** and **textArrows** set? Experiment with **lockScreen**. Be sure you understand that *HyperCard* automatically resets its value when your script is finished. Try changing the pointer shape using the **cursor** property. The "busy" pointer generally rotates when you see it in other Macintosh applications. You can cause that to happen by writing something like

```
on mouseUp
   set cursor to busy
   wait 1 second
   set cursor to busy
   wait 1 second
   set cursor to busy
   wait 1 second
   set cursor to busy
end mouseUp
```

Try modifying this script until you get an effect you like.

Chapter 9. Design Issues

In this chapter, you will focus on the design of *HyperCard* stacks. In particular, you will learn about stack structure, appropriate use of visual effects, and some *HyperCard* conventions you should take into consideration when designing stacks.

The first eight chapters of this book have focused on the mechanics of writing commands in HyperTalk. You have learned how to write scripts for buttons and fields. You have learned how to use visual effects. You have learned how to use the Message box to write HyperTalk statements, about the properties of buttons, fields, and windows, and how functions work. You have also taken a look at the structure of both *HyperCard* and the HyperTalk programming environment. Much of your work has been experimental, not producing any particular product.

However, as you learn more about scripting in HyperTalk, you will want to begin to apply your ideas to building stacks that are of value to you and to others. *HyperCard* is more than a programming environment. It is a powerful tool for creating a graphical interface to work with information contained in stacks. *HyperCard* is designed to make the creation of stacks quick and easy. However, it is also quite easy to produce very badly designed stacks. The user of such stacks can easily become confused or lost. Paying careful attention to the design of stacks is as important as the mechanics of writing scripts for a stack.

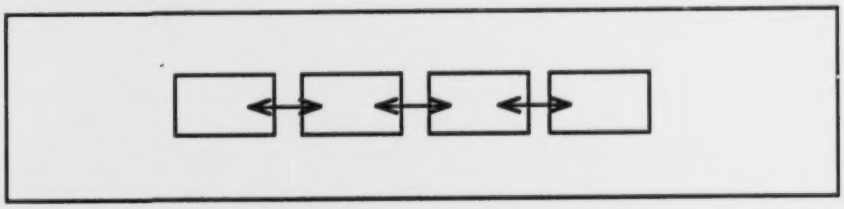
Stack Structure

The first step in creating a *HyperCard* stack should be planning the structure of the stack. There are many ways you can put a stack together. One effective approach to stack design is to "outline" your stack on index cards. Using this method, you lay the cards out on a table to help you decide upon the links among cards. You might also consider using a bulletin board to arrange the cards in your stack. You then use pieces of string or yarn to indicate links among cards. This approach lets you take a look at the overall structure of your stack and think through what relationships among cards in your stack are important.

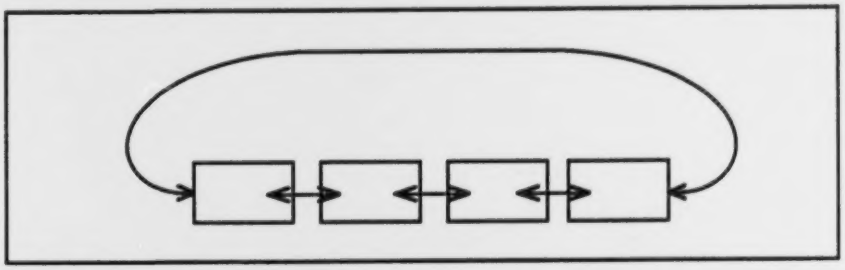
When designing a stack it is extremely important that you think carefully about the nature of the material you want to put into a stack. Is it "linear" in nature? That is, do you want your readers to see one card at a time, much as if they were reading a book? Or does the material "branch" naturally. Is it sensible for the user of your stack to want to explore one area for a while—such as the habitats of animals—and then be able to move with ease to the foods eaten by those same animals? Is your stack exploratory? That is, do you want users to be able to explore in a variety of ways and jump from place to place among the cards in your stack in no particular order? Answering questions such as these requires a lot of planning, examination of the material that is to become part of your stack, and thought about the nature of the final product.

There are some general formats stacks can take. These are discussed in the following sections. However, it is important that you *not* take these as rules but rather as general ideas about ways to structure stacks. You may create stacks that are exactly like some of the following models, or stacks that are totally different from these models. Or you might choose to use some combination of these ideas. The structure of your stack depends heavily on the nature of the material that makes up your stack.

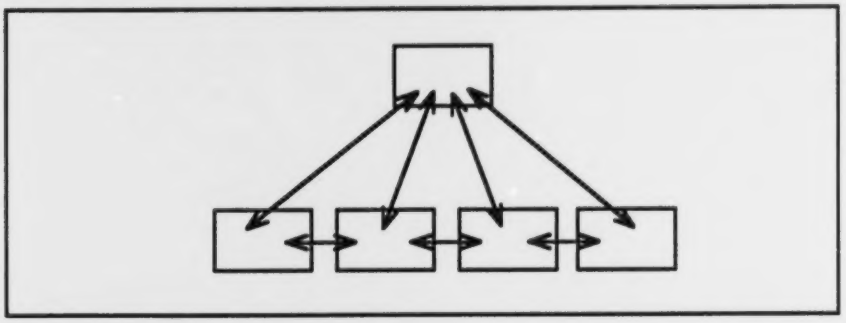
Stacks may be strictly linear. You might use this type of stack if it is important that the user of your stack see each card in sequence. This structure might be used when you have a sequence of instructions and you want to be sure the user of your stack sees each instruction in order:



It is not uncommon for a linear stack to add a link between the first card and the last:

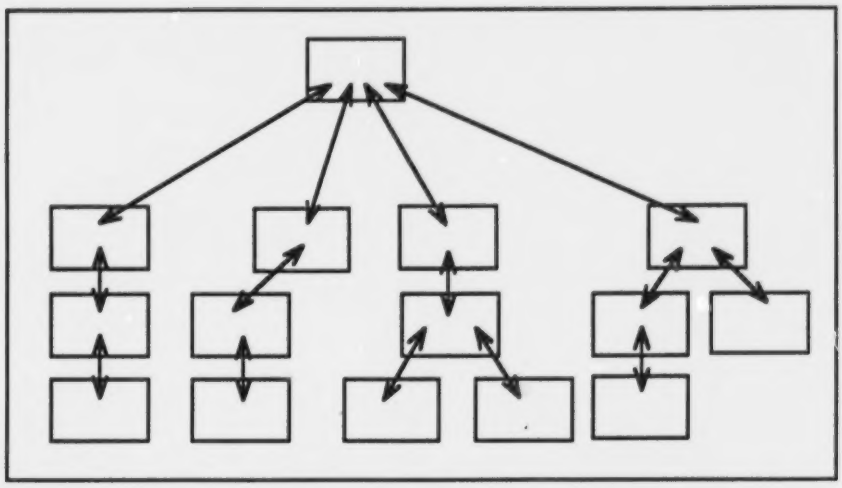


You might also want to make your stack "jump linear," allowing the user to jump into or out of any card in the linear part of the stack:



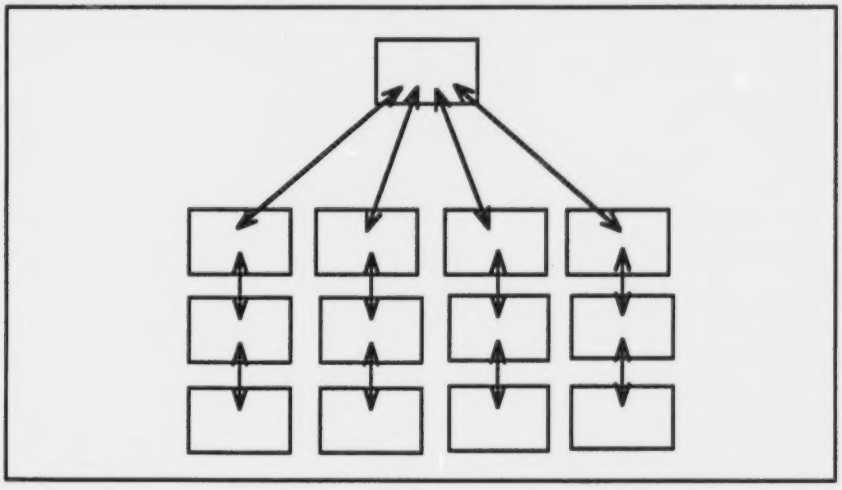
The jump linear model might be used to present a menu on the first card—the top box in the preceding diagram. The user can then go to any card in the linear sequence. Once in the linear sequence, the user can either go back to the menu or move back and forth in the linear sequence.

Computer scientists use the term "tree" to describe the structure shown in the following diagram:

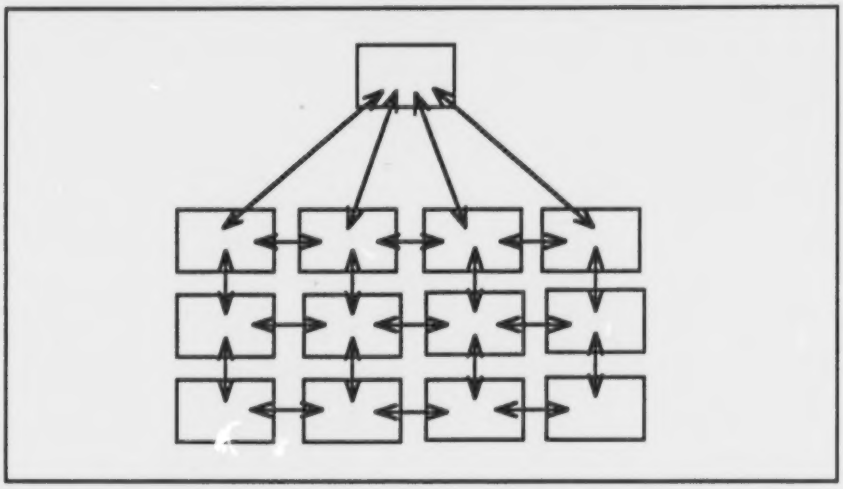


A tree structure allows users to move "up" and "down" branches of the tree to explore any particular idea in more depth. Tree structures are fairly natural ways to display information. For example, if your stack is about animals, you might have two subbranches: water animals and land animals. Then, under water animals, you might have other divisions, such as vertebrates and invertebrates.

There is a temptation when working with tree structures to make as many links as possible among cards. Such an approach can be quite confusing to the user. The following tree structure is fairly easy to follow. Users know that, once they are in a given branch of the tree, they will remain in that branch until they return to the top of the tree.

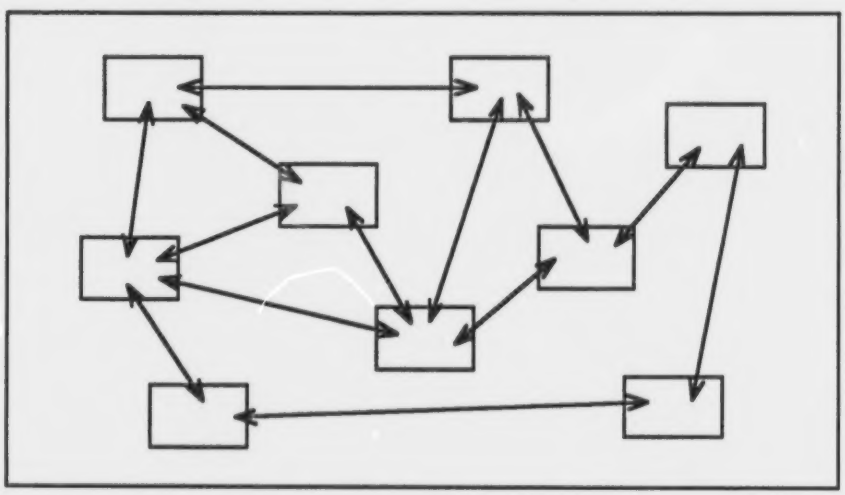


However, if many links are made that allow movement across the branches of the tree, users can easily become lost. The following diagram shows a structure in which it is easy to become lost:



In this case, if users go down the "water animals" branch of such a tree, and then move to the "right" or "left" in the tree, they can easily find themselves in a "land animals" branch without being aware that the nature of the information being presented has changed. Making as many links as possible does not necessarily make your stack better. However, you need to be sure the links to the branches of a tree are bidirectional; that is, be sure the cards are linked so that a user can go "down" and then back "up" a given branch of the tree in some manner.

Finally, network stacks are used when you want the user to explore your stack in a variety of different ways. Such stacks present information that has no particular order to it. For example, you might represent airline routes in such a stack. Or you might create an exploratory stack for children so that when they click on a picture of an object, they move to a related picture.



Although the model that we use to think about *HyperCard* stacks is a deck of index cards, not all stacks appear to have a lot of cards. "Single-frame" stacks appear to have only one card. The user has no sense of moving through a number of cards at all. One type of single-frame stack is the "display stack." This type of stack is used to show a graphic or text field that changes depending on the user's choice. For example, you might create an alphabet stack for children. Along the top of the card there would be 26 buttons, one for each letter of the alphabet. When the child clicks on the letter, a picture of an object beginning with that letter appears. In such a stack, the pictures might appear in a standard "frame" on the card and no visual effect would be used to create a sense of movement.

Another form of single-frame stack is the "filter stack." This type of stack may serve as a kind of menu for a single stack or "front end" for another group of stacks. Filter stacks are particularly useful when a stack becomes very large and contains a lot of information. For example, a stack that contains information about all of the major countries of the world might filter the information by allowing the user to choose to look at, for example, African countries only.

A third type of single-frame stack is a "point-and-click" stack. Such stacks let the user click on various items on the card to learn more about them. For example, a stack that presents a map of a city might give brief descriptions, addresses, and phone numbers of restaurants when the user clicks on the appropriate picture on the map. In such a single-frame environment, the user would have no sense of moving to another card; rather, the information would appear to "pop up" on the original map card.

Using Visual Effects

Visual effects are a lot of fun. It's quite easy to end up using visual effects that are inappropriate for your stack just because *you* enjoy looking at them. Perhaps you have already discovered how easy it is to over-use visual effects as you were developing a stack. After you see the same powerful effect many times as you are debugging, you soon find that these effects can be irritating as well as exciting.

You should use the **visual** command only when you have a specific reason for doing so. It is impossible to give a set of rules for using visual effects. What you see on the screen can be rather subjective. An effect that appeals to you may annoy another person. Furthermore, the effect that you see on the screen is strongly influenced by the cards involved. A visual effect that is quite strong in the case where the first card is quite different from the second can become quite weak when the first card is almost like the second. Thus, the following guidelines are very general.

Scroll

As you know, a scroll makes the next card appear to slide into position from the side or top. From the user's point of view, the "new" card does not appear to have been underneath the first. This effect generally makes users feel as if they are moving along to a new topic. This gives the feeling that a transition of some kind has taken place. Scrolls are also good when you want to simulate a slide show, because the effect on the screen is quite similar to seeing a set of slides projected by a slide projector.

Dissolve

Dissolves give an emphasis to the transition itself. You should use a dissolve when there is a significant graphic contrast between cards or when you are changing one small part of a card and want to draw the attention of the user to that change. If the cards are

quite different, a dissolve can give the user time to adjust. Dissolves tend to be midway between scrolls and wipes in strength.

Wipe

A wipe gives more of an impression that a page has been turned and that the next card is underneath. The transition produced by a wipe is less abrupt than a scroll. This is a good effect to use if you want users to feel that they are continuing at the same level.

Iris/Zoom

These two effects are quite powerful. They can simulate the feeling of going into or out of three-dimensional space. They give users a feeling of depth, as if they are going shallower or deeper into the material being presented.

Barn Door

This effect produces a feeling of finality, like opening or closing a scene. There is no sense of transition as the user moves to the next card. This effect should be used to indicate the beginning or end of a major section of a stack.

Checkerboard/Venetian Blinds

These two effects are very powerful. They should be used only for a strong visual break and should be used sparingly. Many users find them annoying, especially when used with "to black."

HyperCard 2.0 Effects

The *HyperCard* 2.0 effects **shrink** and **stretch** are similar to scrolls when used with "to top" and "to bottom." They are more like irises and zooms when used with "to center." However, in both cases these two effects are somewhat more dramatic than their counterparts.

Combination Effects

You will also want to use combination effects at times. For example, if you feel that the iris effect alone isn't strong enough, you might try

```
visual effect iris close to black
visual effect iris open
```

This gives the impression of an iris closing, leaving you briefly in the dark, and then reopening, leaving you at a new card. This is a much stronger transition that just **iris open** or **iris close.**

Another strong effect might be

```
visual effect zoom open
visual effect dissolve fast to white
visual effect dissolve fast
```

This gives an almost explosive effect. This is the sort of effect that you want to avoid over-using.

Another interesting effect is to go to the same card with the first of several visual effects. Thus, you might use

```
visual effect scroll left
go to this card
visual effect dissolve slowly
go to next card
```

The second line causes the card to go to itself, so you see it "scroll left" to itself and then you see a dissolve as it goes to the next card in the stack.

Perhaps you would like to simulate the black transition as you go from slide to slide in a slide show. How about something like

```
visual scroll left very fast to black
go to next card
visual scroll right very fast to black
go to this card
```

You would probably need to adjust the speed, depending on the effect you want and the speed of the Macintosh on which you are working.

Note also that when you use an effect such as **iris open** to go to a card, you will likely want to use **iris close** when you reverse your path to come back to the original card. That is, if you want users to feel as if they are "going deeper" as they move to the next card, then you want them to feel as if they are "going shallower" when they come back.

Be very careful not to overdo visual effects—especially combined visual effects. You should use **visual** when there is a good reason for doing so, not just because it is "pretty" or "fun." Consider the following analogy. Visual effects are like doorways. You put a door (visual effect) in only when you want to make a transition explicit or close off one section from another. You don't put a door between your chair and your television, because it would be in the way. Nor do you put a heavy bank-vault door on your kitchen cabinet when an easy-to-open door serves the purpose much better.

Backgrounds

Some beginning users of *HyperCard* seem to use the same background all the time. They simply take the background *HyperCard* puts into a new stack and use it for every stack they create. Others, however, discover that they can put any kind of picture on the background. These people end up with a different background on every card. Yet others put "next" and "previous" buttons on every card, forgetting that they can simply put those buttons on the background and they will be available on every card that they create that uses that same background. None of these approaches makes appropriate use of *HyperCard* backgrounds.

The initial background you choose for your stack should reflect the content of your stack. It certainly is inappropriate to have a background that looks like index cards if your stack is supposed to look like a slide show. You should give a good deal of thought to the backgrounds you design for your stack.

Because it is so easy to edit and change backgrounds, it is tempting to have a lot of backgrounds in your stack. This can be very confusing to the user of your stack. A background change should represent a change in the material being represented in your stack. For example, if you are teaching about animals and plants in your stack, you should have at least two backgrounds: one to indicate that the user is in the section about animals and one to indicate that the user is in the section about plants.

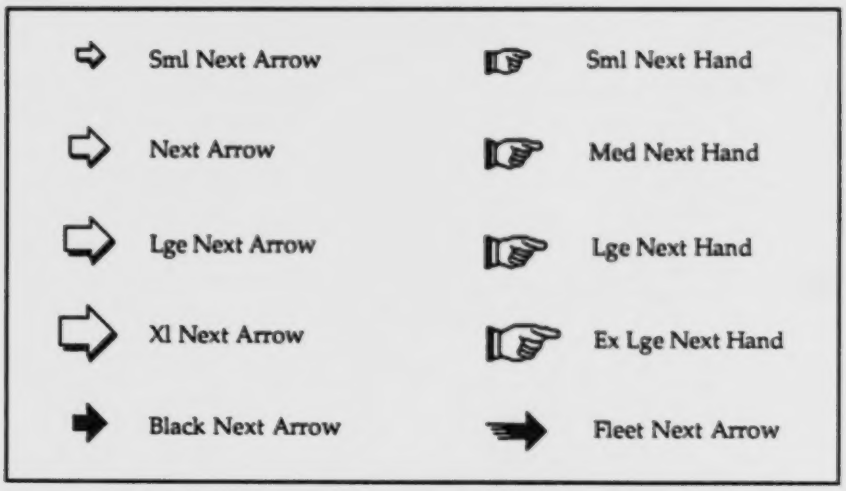
Yet another problem with multiple backgrounds is that they consume a lot of disk space. Cutting down on the number of backgrounds will reduce the size of your stack.

On the other hand, you should make use of the power of backgrounds. If you have a button or field that needs to appear on every card, put it on the background. If there are just a couple of cards that don't use that particular button or field, you can cover the background objects with invisible buttons when needed. Think about redesigning your background if you find yourself putting the same object on card after card.

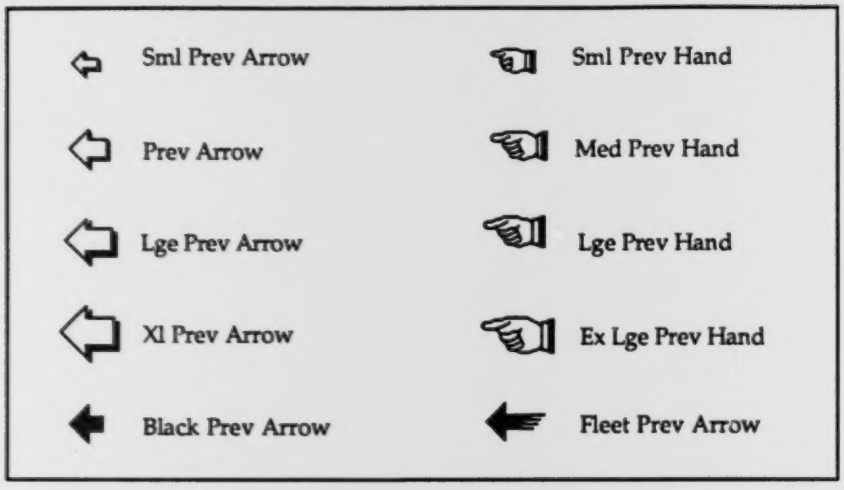
Finally, designing your background is as important as designing the structure of your stack. Carefully thinking about what should be on the backgrounds in your stack in advance of developing your stack will save you a lot of time at the computer keyboard.

Conventions for Using Icons

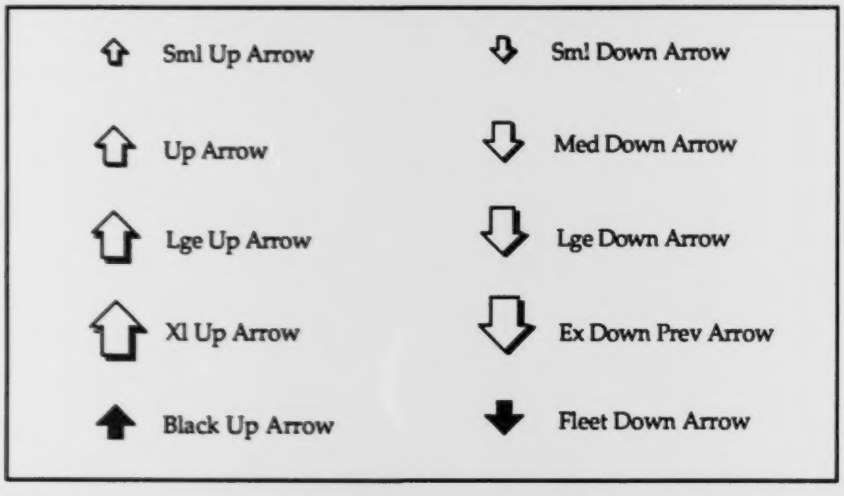
There are some conventions for using various icons on buttons. If you deviate too much from these conventions, users who have used other *HyperCard* stacks may have a great deal of difficulty getting around in your stack. For example, the following icons standardly mean "go to the next card" in the stack:



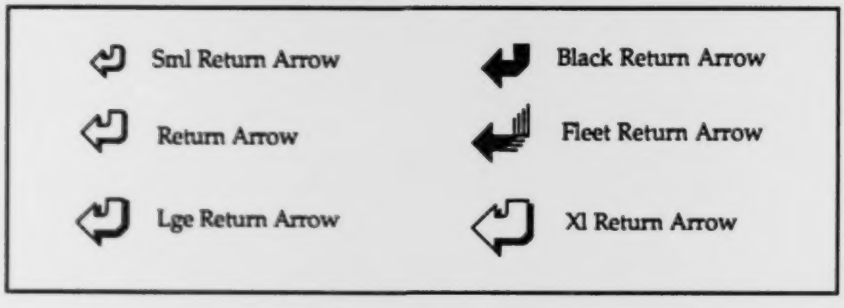
On the other hand, the next group of icons usually mean "go to the previous card":



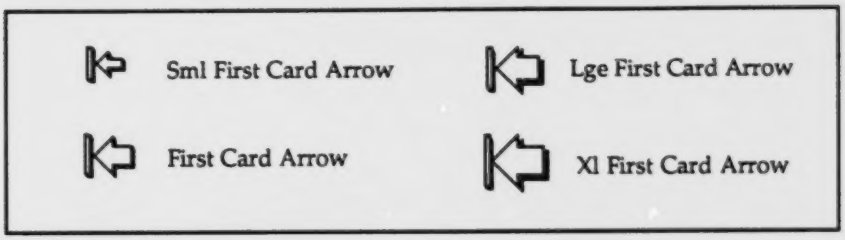
In addition, the following "up" and "down" arrows generally mean go to the previous card and go to the next card, respectively:



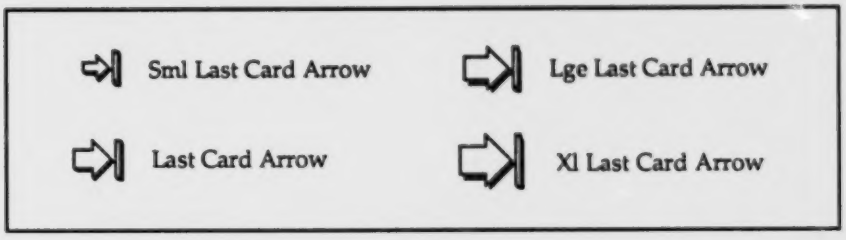
The following set of icons commonly means "return." Return can have a lot of different meanings in *HyperCard*. Return can mean to go back to the "menu," to return to the last "branch" in the stack, and so forth.



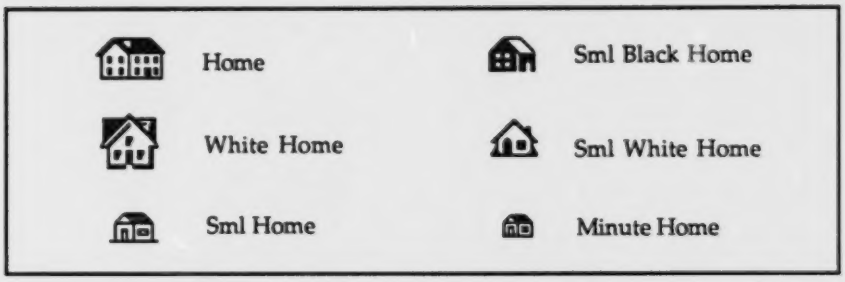
This next set of icons generally means to go to the first card in the stack,



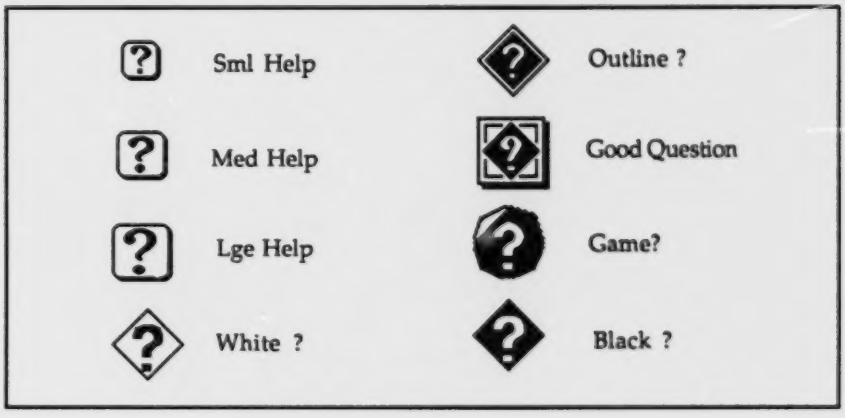
while the following icons generally mean to go to the last card in the stack:



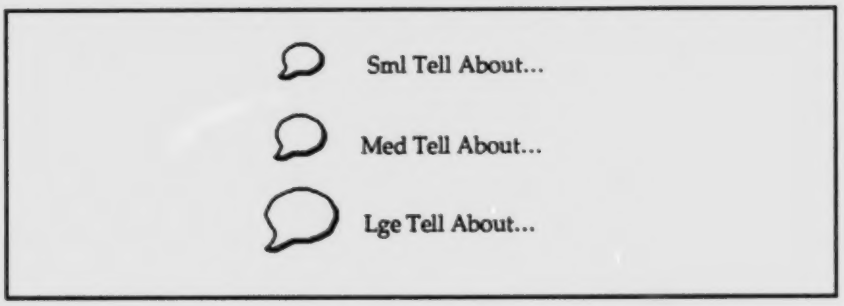
An icon that looks like a small house almost always means to go to the Home card:



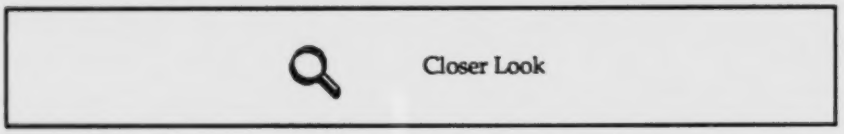
Similarly, a question-mark icon generally means that the user should click there for help:



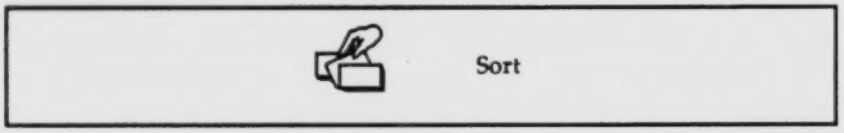
Little "balloons" generally mean "tell me about,"



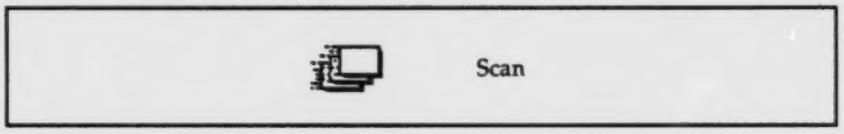
while the magnifying glass means to take a closer look, or perhaps "find,"



Finally, there is the icon that generally stands for "sort." This icon is used when the user wants to order material in the stack in some way:



There is also an icon, which you saw earlier in this book, that generally means to show all of the cards in the stack:



Some Other Considerations

It is a good idea to include a Home button on every card in your stack, or at least on cards that are easily reached at any time when using your stack. It is very easy for novice users to become confused when using *HyperCard*. Allowing them to return to the Home card gives them a way to start over.

Similarly, you should think carefully before hiding the Menu bar. If you expect your stack to be completely self-contained or to be used only by experts, there is no problem. However, new *HyperCard* users may not know that they can press Command-Space bar to show and hide the Menu bar. Even a small problem in stack design can be very frustrating for beginners who finds themselves without access to the standard Macintosh menus.

You should think carefully about the user level you want built into your stack. An experienced *HyperCard* user will be able to modify the user level from the Message box, but a novice will not. If it is important that the user not be able to make any changes in your stack, set the user level quite low. If you want to allow the user to modify the

stack, set the user level higher. You can add a script at the stack level (see Chapter 18) that will set the user level automatically.

Another important consideration when designing stacks is consistency. If buttons and fields are not placed carefully, the transitions between cards will be unpleasant. Buttons and fields may appear to jump around if they are not at exactly the same spot on each card. You can avoid that problem by using copy and paste. A button copied from one card will automatically be pasted in exactly the same spot on another card.

Even worse than having buttons appear to shift slightly as you move through the stack is having buttons that accomplish the same task appearing at different places on every card. This forces users to search for buttons they need on each card.

If you use icons on buttons, make every effort to be sure that the meaning of the icon is clear. Even better, put both the icon and a word on buttons to make their functions clear.

Consistency and clarity in placement and use of the various elements of your stack are quite important. Keep in mind that *you* are very familiar with the design of your stack. A new user is not. You must be sure that everything on your stack is easy to understand.

Debugging and Testing

When using visual effects, you need to think carefully about the final effect you want. For example, if you want to use a **scroll** effect to go to the next card, it is likely that you will think of the next card as being "to the right." For example

```
visual effect scroll right
```

makes the next card appear to come from the left. You must use

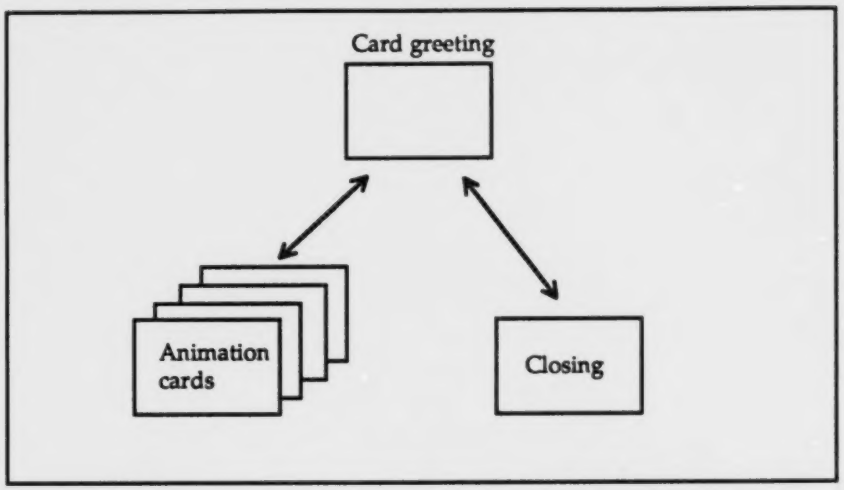
```
visual effect scroll left
```

to have the next card appear "to the right." Be sure you try your visual effects and think carefully about the actual effect you want to achieve.

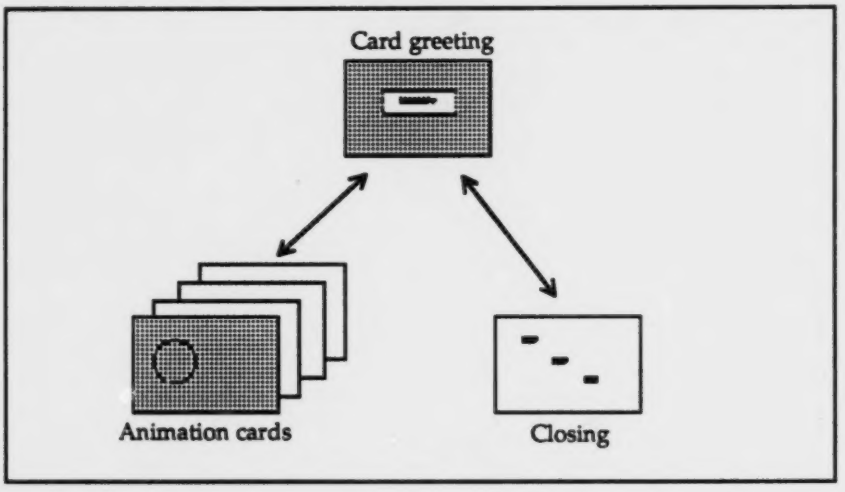
Visual effects don't work if the computer monitor being used is set to color. If your stack is dependent on visual effects, you may want to tell the user of your stack to be sure to turn the color off before using the stack.

The speed of visual effects varies depending on the machine you are using. If you develop your stack on a Macintosh Plus and then run it on a Macintosh SE/30, you will see a great difference in the speed of your visual effects. Be sure to design your stacks for the machines on which they will eventually run.

It is easy to lose track of the structure of your stack as you are working, even if you planned it carefully in advance. To help visualize your stack, it is a good idea to make a stack map. A stack map looks like the diagrams at the beginning of this chapter, but with individual cards identified. For example, you might have



HyperCard provides a method for making stack maps so that they can easily become a part of your stack, perhaps serving as a guide for the users of your stack. You can make "tiny cards"—miniature copies of the actual cards in your stack. The preceding stack map might look like this:



To make these small card images,

1. Select **Copy Card** from the **Edit** menu.
2. Move to the card on which you want your map.
3. Press Command-Shift-V.
4. Now move the image into place.

You can then use the Paint tools to draw arrows or add other details.

Project Ideas

Many of the ideas briefly summarized in this chapter are discussed more extensively in a variety of *HyperCard* books. One particularly good source of design

information is *HyperCard Stack Design Guidelines,* published by Addison Wesley and produced by Apple Computer, Inc. Spend some time with a book such as this one, and think more about the issues involved in creating good stacks.

Experiment with different combinations of visual effects. Keep a record of the ideas you try that you might want for later use.

Examine some existing stacks. Are they easy to understand? Why or why not? See what kinds of overall stack design you can find and think about the ones that would be most useful to you.

Chapter 10. More on Button and Field Properties

In this chapter you will learn more properties that you can use with both buttons and fields.

Earlier in this book, you learned that you could use HyperTalk to set or change the button and field properties found in the **Button Info** and **Field Info** dialog boxes either by using the Message box or by using handlers. You also learned that there are properties like **location** and **rectangle** that you can change by using HyperTalk but that you can't change by using the dialog boxes under the **Objects** menu. In this chapter, you will learn to use other properties that can only be changed by using HyperTalk.

Using Visible

Recall that the Tool window, Pattern window, Card window, and Message box have a **visible** property. You can also use **visible** with buttons and fields. For example, if you type

```
set the visible of card button 1 to false
```

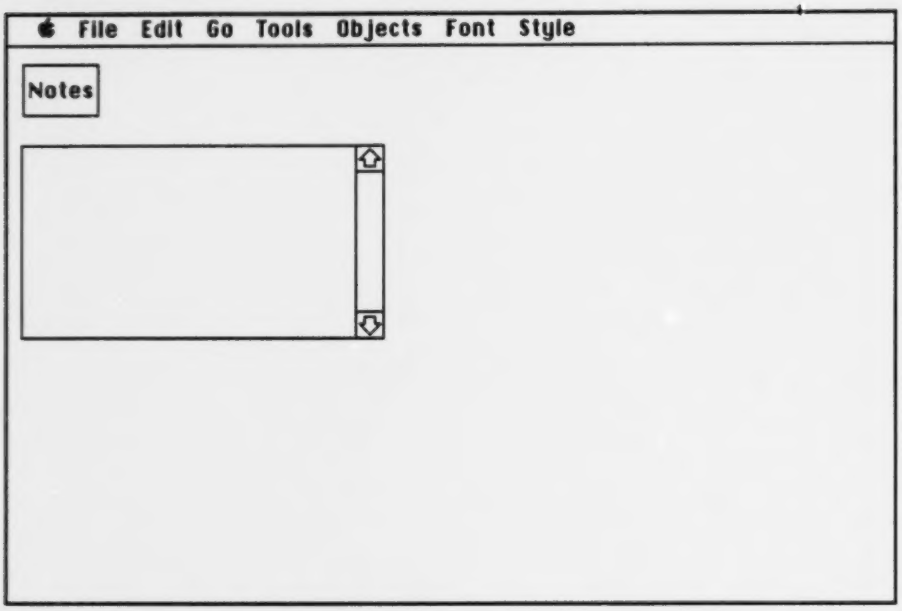
or

```
set the visible of card field 1 to false
```

these two objects cannot be seen by the user. When **visible** is **false**, clicking over the button location or trying to type into the field simply doesn't work. You must reset **visible** to **true** in order to use these objects. Recall that setting **visible** to **true** has the same effect as using the command **show**. You can use either **show** or **hide** to make buttons and fields visible or invisible, or you can set the **visible** property to **true** or **false**.

You might use **visible** to create a pop-up field on which the user can keep notes on each card. Begin by creating a new background from the **Objects** menu. Put a "Notes" button on the background so that it will appear on every card with this background. Next, create a card field called "Card Notes." (If you put the field on the background, then when it appears, it will be "under" everything else on the card. See the diagram

in Chapter 0 that shows the "layering" of *HyperCard* objects.) Your card will look something like this:



Next, put the following script on the button:

```
on mouseUp
   set the visible of card field "Card Notes" to true
end mouseUp
```

Then put this script on the field:

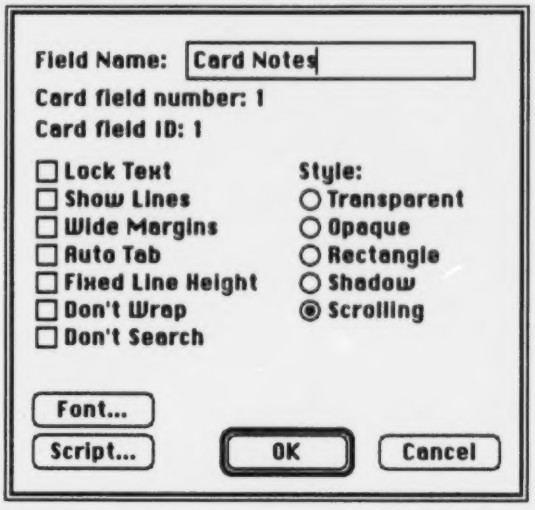
```
on closeField
   set the visible of card field "Card Notes" to false
end closeField
```

If you now click on the button, your Card Notes field appears. Type some information on the field, and click outside the field. The field will disappear. Next time you want to make more notes, click on the Notes button to again see your field. Now you can make a number of copies of your card so that each card in your stack will allow you to add some personal notes to it.

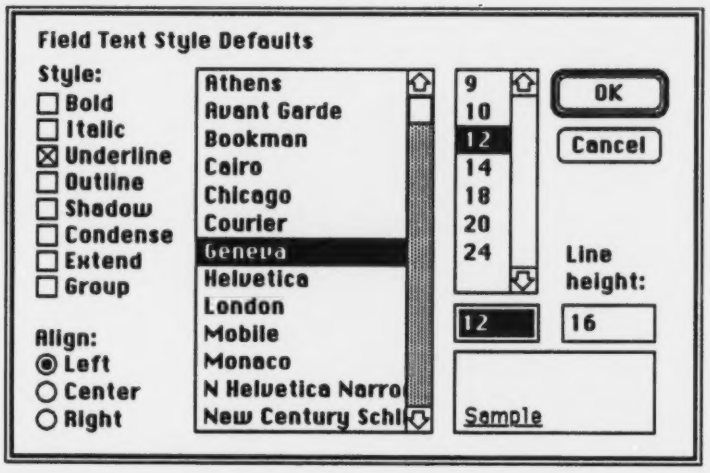
Technical Note: The **closeField** *message is sent only when you have typed something into the field. If you don't type anything in the field, you won't get the* **closeField** *message when you click somewhere else. However, if you click on a line below any text in a field,* HyperCard *will insert Return characters for you. For example, if you simply click on the fifth line of a field containing two lines of text,* HyperCard *will automatically insert three Return characters for you. In this case,* HyperCard *behaves as if you are "typing" something into the field so that after you click somewhere else, you will get a* **closeField** *message.*

Working With Fonts in Fields

Recall that the **Field Info** dialog box has a **Font** button:



If you click on this button, you see



Using this dialog box, you can choose the font from those available to you in the current Macintosh System, choose the size and style of the font, choose the line height, and select the text alignment. As you might guess, each of these selections can be changed by using HyperTalk commands. That is, you can use the following field properties:

```
textStyle
textAlign
textFont
textSize
textHeight
```

The following HyperTalk commands will produce the same effect as selecting them in the dialog box in the previous illustration:

```
set the textStyle of card field "Card Notes" to underline
set the textAlign of card field "Card Notes" to left
set the textFont of card field "Card Notes" to "Geneva"
set the textSize of card field "Card Notes" to 12
set the textHeight of card field "Card Notes" to 16
```

The possible values of **textStyle** are those given in the previously shown dialog box. Note that you can use more than one style at a time:

```
set the textStyle of card field "Card Notes" to bold,
   underline, extend
```

The names of the styles must be separated by commas.

Technical Note: If you use a combination of text styles along with **plain**, *all your text will be plain style. That is,* **plain** *implies that the text style is completely normal, so text cannot be both bold and plain at the same time. Note also that if you are using* HyperCard *2.0, you can use multiple fonts in one field. In addition, there is a text style* **group** *that allows any group of characters, including more than one word, to be selected at a time.*

TextAlign determines the way the text is placed on the line. Text may be centered, left aligned, or right aligned. You use the words **left, right,** and **center** to refer to the alignment of text. The **textFont** may be any of the fonts available to you. If you use a font not in your set of resources, *HyperCard* defaults to Geneva.

The size of the font you wish to use is set by using **textSize**. If you select only sizes that are defined in your system, the quality of the text that appears in your stack will be better. You can tell which sizes you have available by examining the dialog box. The numbers that appear just to the left of the **OK** button are the sizes defined in your system. You can, however, actually choose any size you want, and your Macintosh will create an approximation of that size.

The **textHeight** property determines the space between lines of text. **TextHeight** defaults to a number that is approximately the **textSize** plus one-third of **textSize.**

You can make use of these properties to examine any field in your stack. For example, you can type such commands as

```
put the textSize of card field "Card Notes" into the Message
   box
```

to determine the size of the current text in the field. Recall that this can be shortened to

```
the textSize of card field "Card Notes"
```

since the Message box is the default container for property values.

Working With Fonts on Buttons

All the buttons you have created have the same font (Chicago) in the same size (12 point). There is no **Font** dialog box in the **Button Info** box. However, it is possible to use the same text properties with buttons that were described previously for fields.

Go to your card with the Notes button on it. Then type

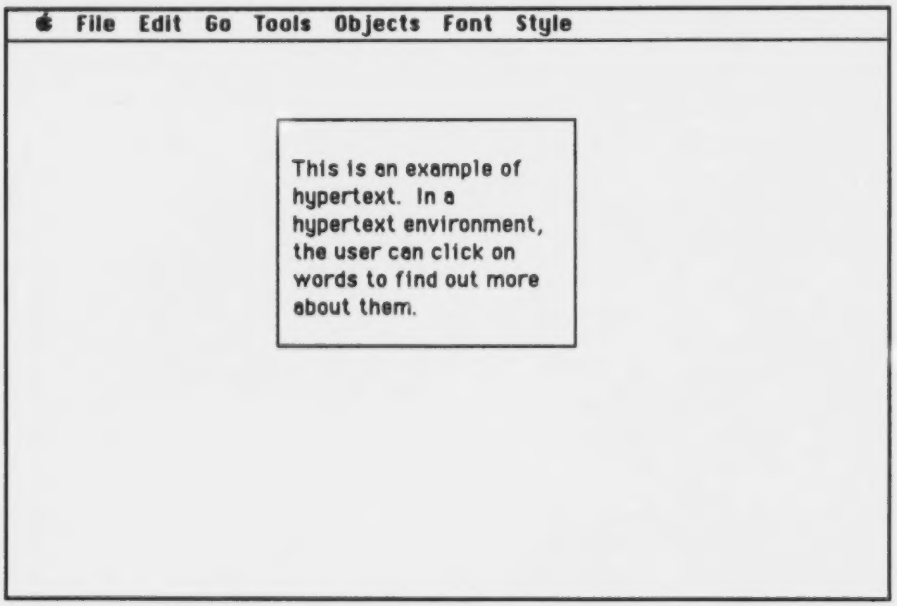
```
the textFont of card button "Notes"
the textSize of card button "Notes"
the textHeight of card button "Notes"
the textStyle of card button "Notes"
the textAlign of card button "Notes"
```

You should see the following values appear in the Message box:

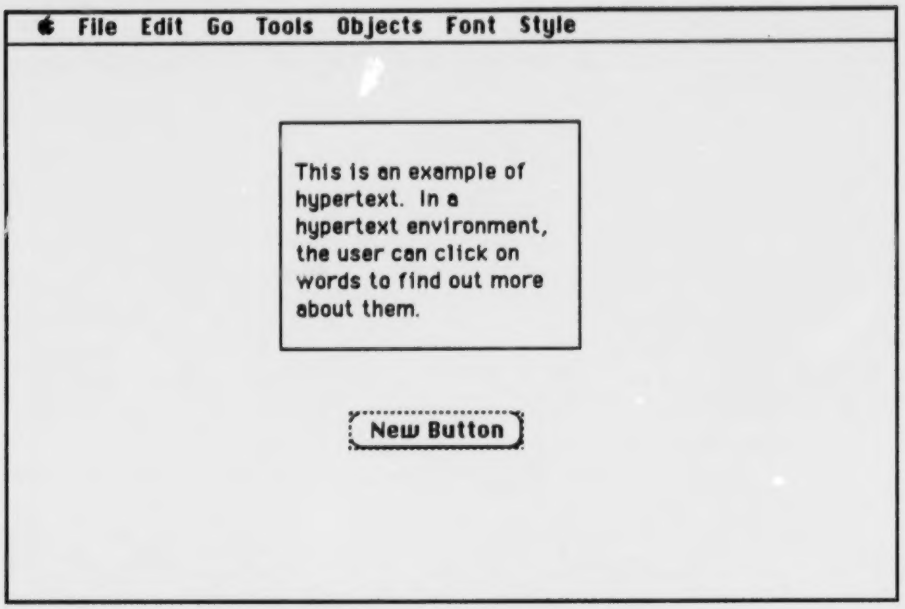
```
Chicago
12
16
plain
center
```

Technical Notes: Although the ***textHeight*** *property exists for buttons, it has very little meaning. Text height determines the spacing between lines of text, but buttons have only one line of text. Thus, there can be no spacing between lines.*

The ability to change the text on a button is one technique you can use to create hypertext. Start with a card that has a field containing some text:


File Edit Go Tools Objects Font Style
This is an example of hypertext. In a hypertext environment, the user can click on words to find out more about them.

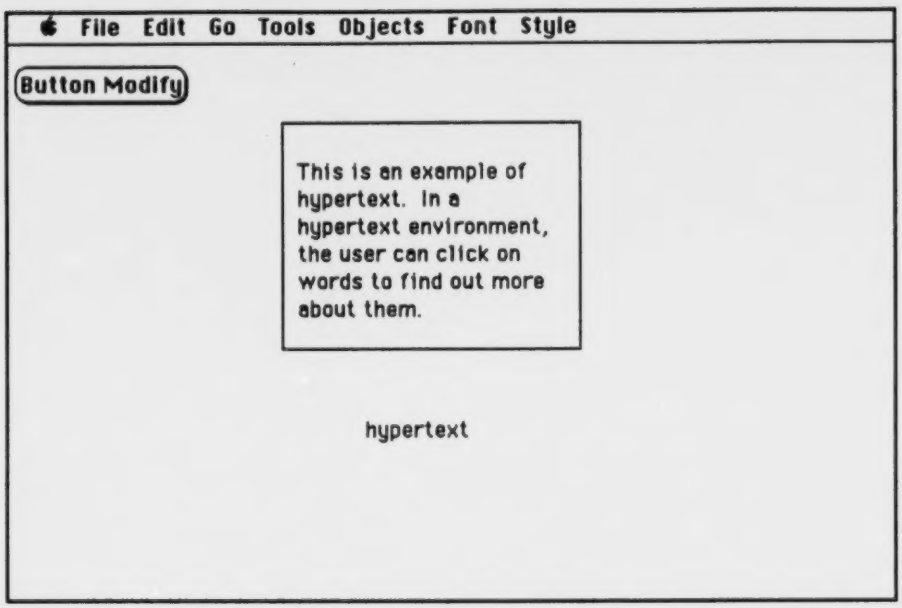
Next, create a new button:


File Edit Go Tools Objects Font Style
This is an example of hypertext. In a hypertext environment, the user can click on words to find out more about them.
New Button

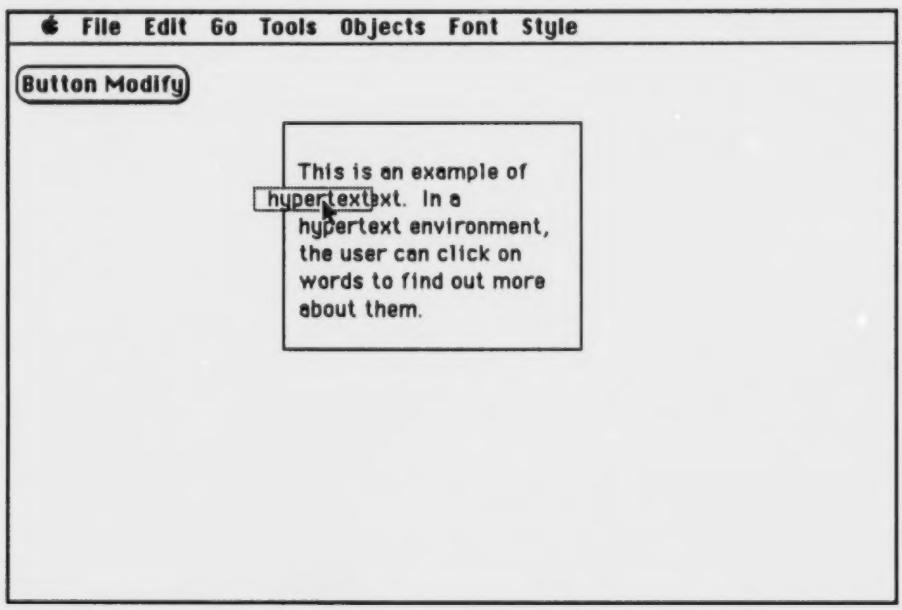
Use the following commands to change the text on the button:

```
set the style of card button "New Button" to opaque
set the textFont of card button "New Button" to "Geneva"
set the textSize of card button "New Button" to 12
set the textAlign of card button "New Button" to left
set the name of card button "New Button" to "hypertext"
```

You can type these commands one at a time in the Message box. However, it is often easier to create a Button Modify button and then write a handler for Button Modify that includes the appropriate statements to change the text on your new button. Clicking on the Button Modify button then makes all your changes at once. If you want to make additional changes, you don't have to retype a lot of lines. Now your card might look like this:


File Edit Go Tools Objects Font Style
Button Modify
This is an example of hypertext. In a hypertext environment, the user can click on words to find out more about them.
hypertext

Next, be sure that the button fits tightly around the word. Drag it over the matching word on the text:


File Edit Go Tools Objects Font Style
Button Modify
This is an example of hypertext. In a hypertext environment, the user can click on words to find out more about them.

Next, add a handler to the "hypertext" button, such as

```
on mouseUp
   go to Card "Explain Hypertext"
end mouseUp
```

When you are finished you will find that clicking on this button will give you more information about hypertext. You have now created the beginnings of a hypertext environment.

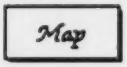
Another technique for creating hypertext is to use transparent buttons placed on top of fields so that the words under the buttons appear to highlight when you click on them. Those buttons then contain handlers that link the text in the field with other cards. If you are using *HyperCard* 2.0, you can use the Group text style to create hypertext as well (see Chapter 19).

The ability to create buttons with a variety of type styles has value far beyond creating hypertext. A variety of type styles gives you more flexibility in designing your user interface. For example, suppose you have a button that links to a map:

Perhaps all of the rest of your card uses a script font and you want your button to match the rest of the card. You can type

```
set the textFont of card button "Map" to "Zapf Chancery"
```

to modify the button so that it looks like the type on the rest of the card:



Debugging and Testing

As you are working with *HyperCard*, you must pay attention to the level on which buttons and fields are placed. You know that card buttons and fields are always "on top of" background buttons and fields. That is, they are at a higher "level." However, each time you create a new card button, that button will be "on top of" the other buttons and fields you have created on that card. The same is true of card fields. Similarly, background buttons and fields "pile up" as they are created, with the one created most recently being "on top." You can check the level of these objects by moving them around with the mouse. If, for example, the button goes behind a field when moving across that field, it means the button is at a lower level than the field. On the other hand, if it stays on top of the field, the button is at a higher level.

In the hypertext example just described, you need to be sure to create the "hypertext" button after you create the field. However, you can change the level of

buttons and fields after they are created by using the **Objects** menu (or Command– and Command–+) to send particular objects further back or bring them further forward:

As you move buttons and fields, their numbers may change. That is, the button that is "lowest" on the card will be numbered "1," the next one on top will be numbered "2," and so forth. Note, however, that you will not be able to move background fields or buttons in front of card fields or buttons. Card buttons and fields are always "above" background fields and buttons.

New HyperTalk Words in This Chapter

Properties	
`textAlign`	`textSize`
`textFont`	`textStyle`
`textHeight`	`visible`

Keyboard Shortcuts	
Command-+	Bring an object closer
Command--	Send an object farther away

Project Ideas

Experiment with different text in fields and on buttons. Which style and font would you use for what purposes? Think about how you might use the font size or style to interact with users in a more meaningful way.

Create a hypertext environment. Link all the important words in a brief passage of text to cards that explain those words. What about linking words on the "second level" to yet other cards at a deeper level or at the same level? Think about the implications of building links such as these.

Create a miniature word processor. Make a card that contains a rather large text field and several buttons. These buttons should allow you to change the font, centering, or style of the text in the field.

Personalize the example of the pop-up field given at the beginning of the chapter. What kind of field would you want to have available for notes? Modify the font, spacing, and so forth to your preference.

Chapter 11. Writing Your Own Message Handlers

In this chapter you will learn how to write your own message handlers. You will also learn how to modularize your HyperTalk code so that it is easier to read and debug.

Since very early in this book, you have worked with message handlers like

```
on mouseUp
   go to next card
end mouseUp
```

or

```
on mouseUp
   beep
   visual effect scroll left
   go to next card
end mouseUp
```

You have seen that all handlers begin with the word **on** and have the word **end** as the first word of the last line. All the handlers you wrote responded to "built-in" messages, such as **mouseUp** or **mouseWithin**. Messages such as these are produced automatically when actions such as pressing the mouse button, pressing a key, or opening a stack occur. Now you are going to learn to write your own message handlers. These handlers have the same structure as the handlers you have worked with so far:

```
on message
   commands go here
end message
```

Notice that the handlers you write must also begin with **on** and **end**. Notice also that you create your own message name. You then insert your own HyperTalk commands just as you have when using the built-in handlers.

Writing your own handlers helps you to write more efficient and easy-to-read code in three important ways:

- You can make your HyperTalk code easier to read.
- You can avoid writing the same commands again and again.
- You can more easily test parts of your code.

The following examples explore each of these ideas.

Making HyperTalk Easier to Read

One of the most important reasons for learning to write your own handlers is to make your HyperTalk code easier to read and understand. One way to accomplish this task is to *modularize* them. Modularizing means breaking your scripts into small, meaningful parts.

Suppose that you want to create a button that plays the song "Row, Row, Row Your Boat" when you click on it. You might write

```
on mouseUp
   play "harpsichord" "cq. cq. cq de eq. eq de eq fe gh. c5e
      c c g4e g g e e e c c c gq fe eq de ch."
end mouseUp
```

The periods in the note list (e.g., "eq.") are used to "dot" the previous duration. For example, the length of a dotted quarter note is one and one-half times as long as a regular quarter note.

If you insert the continuation symbol (¬) into the middle of the quoted line to make the **play** command more readable, it won't work correctly. You must use the concatenate (&) symbol to put the two pieces of the line back together:

```
on mouseUp
   play "harpsichord" "cq. cq. cq de eq. eq de eq fe " & ¬
         "gh. c5e c c g4e g g e e e c c c gq fe eq de ch."
end mouseUp
```

Note that there is a space after "fe" and before the quotation mark, then the &, then the ¬, and then a quotation mark at the beginning of the next line.

While writing long lines and breaking them by using the continuation symbol will work, it is quite hard to read. Furthermore, as you are working with this script, you are likely to find it hard to debug. That is, you will find it difficult to locate and fix any errors you make. Another approach is to modularize your script. To begin, you could write your own handler for the first line of the song in the script editor:

```
on rowRow
   play "harpsichord" "cq. cq. cq de eq."
end rowRow
```

You can then test your **rowRow** handler by inserting it in the familiar **mouseUp** handler:

```
on mouseUp
   rowRow
end mouseUp
```

When you are satisfied with the first line of the song, you can write handlers for each of the other lines, testing them as you go:

```
on gently
   play "harpsichord" "eq de eq fe gh."
end gently

on merrily
   play "harpsichord" "c5e c c g4e g g e e e c c c"
end merrily

on life
   play "harpsichord" "gq fe eq de ch."
end life
```

Then your **mouseUp** handler looks like this:

```
on mouseUp
   rowRow
   gently
   merrily
   life
end mouseUp
```

Notice that by using meaningful names for each handler you write, you make your main handler—in this case, **mouseUp**—much easier to read and understand.

You can also increase the readability of your handlers by using the double hyphen symbol (--). This symbol allows you to insert "comments" into your scripts that *HyperCard* will ignore. Thus, you might write

```
on mouseUp --This handler plays "Row, Row, Row Your Boat"
   rowRow
   gently
   merrily
   life
end mouseUp
```

It is a good idea to put comments in any handler that is more than a few lines long.

> *Technical Note: When you put comments on the same line as a HyperTalk instruction, you must leave a space after the instruction. If the space is omitted,* HyperCard *does not recognize the instruction and your handlers will not run correctly.*

Using Handlers More Than Once

Another reason for modularizing code is that particular handlers can be used more than once. Do you have some favorite visual effects? You might want to write some handlers to name these effects. For example, you might write

```
on blackWhite
   visual effect iris close to black
   visual effect iris open to white
   go to next card
end blackWhite
```

or you might write a handler for a favorite sound:

```
on threeNotes
   play "boing" temp 100 "c4q d e"
end threeNotes
```

These could then be combined into a **mouseUp** handler for your button:

```
on mouseUp
   threeNotes
   blackWhite
end mouseUp
```

These handlers can then be used in other places simply by copying the code. You don't have to rewrite them each time you want to use the same idea. Later, you will learn that you can put such handlers elsewhere in your stack and, for example, access them from a number of different buttons (see Chapter 18). Of course, the longer and more complicated your visual effect or sound, the more sense it makes to write separate handlers for each one. It doesn't save you a great deal of time or energy to substitute a three-line handler for one line of code unless that line is quite complex.

You may also have instances where the same group of commands is used several times in a script. For example, suppose the song you had chosen to attach to the above button was "Three Blind Mice." A number of phrases are repeated in this song. For example, if you write

```
on threeBlind
   play "harpsichord" "eq d ch"
end threeBlind
```

then you can use this particular handler three times in your **mouseUp** handler:

```
on mouseUp
   threeBlind
   threeBlind
   seeHow
   seeHow
   theySheDid
   theySheDid
   threeBlind
end mouseUp
```

You might also consider modularizing complex visual effects. For example, you might write

```
on mouseUp
   --special effects
   doDissolve
   go to this card
   doScroll
   go to next card
end mouseUp

on doDissolve
   visual effect dissolve to black
   visual effect dissolve
end doDissolve

on doScroll
   visual effect scroll left to black
   visual effect scroll right very slowly
end doScroll
```

to make clearer the parts of your special effects.

Testing Your Handlers

A third important reason for learning to write your own handlers has to do with testing. Previously, you learned that you could test each "subhandler" by putting it in the main handler. You can also use the keyword **send** to test handlers. (You will learn more about **send** in Chapter 17.)

Suppose you have a button called "Point to me!" Place it on the card with

```
set the location of card button "Point to me!" to 125,125
```

Add the following handler to the button:

```
on mouseWithin
   set the textStyle of card button "Point to me!" to "bold"
   set the style of card button "Point to me!" to "shadow"
   wait 30 ticks
   set the location of card button "Point to me!" to 140, 140
   wait 30 ticks
   set the location of card button "Point to me!" to 125,125
   set the textStyle of card button "Point to me!" to "plain"
   set the style of card button "Point to me!" to "rectangle"
end mouseWithin
```

This handler is designed to cause the button to draw attention to itself. Any time the mouse pointer touches any part of the button (**mouseWithin**), the handler is activated.

You might break this handler into three smaller, more modular handlers:

```
on newStyle
   set the textStyle of card button "Point to me!" to "bold"
   set the style of card button "Point to me!" to "shadow"
end newStyle

on moveButton
   wait 30 ticks
   set the location of card button "Point to me!" to 140, 140
   wait 30 ticks
end moveButton

on originalStyle
   set the location of card button "Point to me!" to 125,125
   set the textstyle of card button "Point to me!" to "plain"
   set the style of card button "Point to me!" to "rectangle"
end originalStyle
```

Of course, you can put these three handlers into a main handler:

```
on mouseWithin
   newStyle
   moveButton
   originalStyle
end mouseWithin
```

However, there are times that you may want to test an individual handler. To accomplish this, you can type

```
send "newStyle" to card button "Point to me!"
```

in the Message box. You then see if the **newStyle** handler works as expected. This allows you to test each handler individually.

Debugging and Testing

The double hyphen symbol (--) symbol can be useful in the debugging process. By "commenting" lines of your handler that you want *HyperCard* to ignore, you can test particular lines. For example, if you wanted to test the **gently** handler, you would write

```
on mouseUp --This handler plays "Row, Row, Row Your Boat"
   --rowRow
   gently
   --merrily
   --life
end mouseUp
```

HyperCard ignores all the lines preceded by the – symbol—it assumes that a comment follows the – symbol. If you have a handler that just won't work correctly, you can "comment out" all the lines except the first one. Then test the handler. If the first line works, remove the comment symbol from the second line and test the handler again. You can use this technique to locate the line that is causing problems.

Don't forget that you can use Option-Return to insert a "continuation" symbol (¬) in a *HyperCard* line. Breaking long lines into smaller parts can help make your code easier to read and debug. Also remember that you must use the & to "reconnect" long quoted lines broken by the continuation symbol.

Note that *no one* writes bug-free code. Professional programmers spend a high percentage of their time debugging code. You should not be bothered if you spend at least as much time debugging as you do designing and writing code.

New HyperTalk Words in This Chapter

Keyword
send

Special Symbols	
--	to mark comments in code

Project Ideas

Create some simple handlers of your own. A good place to start is with music. Divide the music into individual measures or lines and make each one into a separate handler. If you are not comfortable with music, try working with visual effects.

Look at some of your earlier scripts. Are there places where you could modularize your work by writing your own handlers? Are there places where you could make your work clearer by adding comments? Take some time to "clean up" a couple of your earlier projects.

Create a "library" of visual effect handlers or sound handlers you frequently use. Be sure to give them meaningful names so that when you use them in different scripts you can tell exactly what they do.

Break a long script into several parts. Practice using a variety of methods to debug the script. You can

- insert each subhandler into the main handler one at a time
- use **send** to test each part individually
- use the comment symbol (--) to avoid running particular lines.

Which method do you like best? Which one suits your programming style best? Which method do you think works best in different circumstances?

Chapter 12. Object Descriptors, Constants, Ordinals, The, and Me

This chapter will examine the use of "common" English words in HyperTalk scripts. HyperTalk code often appears to be written in English. However, like any other computer language, there is a limited set of words that may be used and there are particular rules for using those words.

Object Descriptors

By now you know that *HyperCard* makes use of five kinds of objects. You refer to these objects with *object descriptors*. For example, you might refer to card field "First One" or to background button "Try me." More formally, an object descriptor is a combination of the *generic* name of an object and the object's specific designation. The possible generic names are

```
stack
card
background (bkgnd or bg)
button (btn)
field
```

Stacks can only be referenced by their name, but the other *HyperCard* objects can be referenced by their name, number, or ID number. Thus, some legal object descriptors are

```
stack "My Stack"
card 3
field ID 2077
button "My Button"
card button "On Card"
background 4
```

As you have seen previously in this book, you can combine these object descriptors by using **of**:

```
button "My Button" of card "My Card"
field 3 of background ID 3012
button 4 of card "My Card"
```

Special Object Descriptors

There are also some special object descriptors that are used to make *HyperCard* scripts seem more English-like. You have seen such statements as

```
go to next card
```

The word **next** is a special object descriptor. It can be used to refer to the next card or background in the stack. Similarly, you can refer to the card or background before the current one with **previous**.

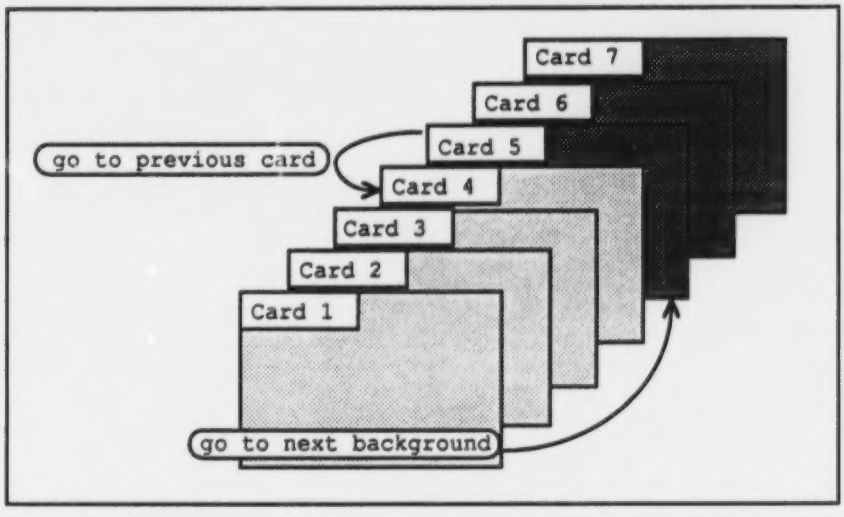
```
go to previous background
```

Note that **previous** can be abbreviated as **prev**.

When you use **next** or **previous** with cards, you move to the adjacent card in the stack. However, if you use these modifiers with *backgrounds*, you move to the first card of the "adjacent" background, which may be many cards distant. That is, if you type

```
go to next background
go to previous card
```

you will likely not be on the card where you began. Suppose your stack looks like this:



If you begin on Card 1 and type

```
go to next background
```

you will be on Card 5. If you then type

```
go to previous card
```

you will then be on Card 4. **Next** and **previous** seem natural and easy to use, but you need to be careful that you are thinking about the physical arrangement of the cards in your stack. If you insert or delete cards from your stack, scripts containing **next** and **previous** may not work correctly. Be careful that you don't write scripts that are so dependent on the order of the cards in your stack that the stack cannot be changed.

Another special object descriptor is the word **this**. You can use **this** to refer to the current card, background, or stack. You might, for example, type

```
put the name of this background into card field "Bg Name"
```

Note that you cannot use **this** with buttons or fields.

Another object descriptor that you may find useful is **recent**. Recall that selecting **Recent** from the **Go** menu allows you to see the most recent cards you have looked at since starting *HyperCard*. The word **recent** can also be used in scripts. Thus, you might type

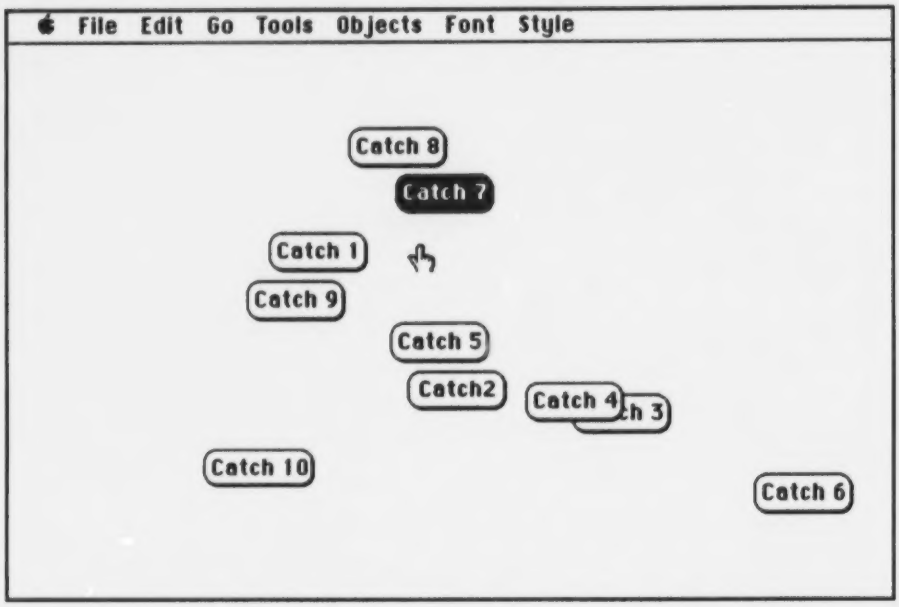
```
go recent card
```

to go to the most recent card you have seen. This command is the same as pressing the Tilda (~/`) key and can be helpful when you are debugging. Typing this command in the Message box lets you step back one card from your current location. You could also put this command into a button on a card to allow users to retrace their paths through the stack.

Yet another English-like word that appears in *HyperCard* is the word **me**. **Me** is used to specify the object of the handler that is currently executing. You might, for example have the following script on a button named Demo:

```
on mouseUp
   put the short name of me into the Message box
end mouseUp
```

When you click on the button, Demo appears in the Message box. **Me** can be quite useful when you want identical scripts on a number of buttons. Suppose you have a series of disappearing buttons. If you touch the button, it highlights, disappears, and reappears elsewhere:



First, however, you need to learn two new functions: **mouseH** and **mouseV**. The syntax of these functions is:

```
the mouseH
mouseH()

the mouseV
mouseV()
```

Try typing one of these functions in the Message box. Then move the mouse. Type the function again. Do you see that **mouseV** returns the vertical (up/down) location of the mouse while **mouseH** returns the horizontal (left/right) position of the mouse? Experiment until you are comfortable with these two functions.

Now, you can put this handler on all your buttons:

```
on mouseUp
   set the hilite of me to true
   wait 1 second
   set the visible of me to false
   set the location of me to the mouseH + 20,¬
      the mouseV + 20
   wait 1 second
   set the hilite of me to false
   set the visible of me to true
end mouseUp
```

Note that the easiest way to accomplish this is to create one button, including the handler, and then make copies of this button, changing the number after Catch on each one.

Examine the handler carefully. Notice that even though the buttons are named Catch 1, Catch 2, and so forth, you can use the same script on each one. Each handler, when placed on the button, refers to the button it is on. Using this technique can save a lot of time when creating a series of identical objects.

Constants

HyperTalk includes a number of *constants*. These constants are words that represent particular values. You have seen a number of these constants in your work with HyperTalk. For example, you have learned that **empty** can be used to represent a string with no characters. You have also seen **return** used to put a return character in a field. You have seen the constants **true** and **false** used with the properties of objects. There is a series of constants that can be used when referring to numbers. These constants include:

`zero`	`four`	`eight`
`one`	`five`	`nine`
`two`	`six`	`ten`
`three`	`seven`	

These can be used in place of the numeric symbols. For example, instead of writing

```
go to card 3
```

you can write

```
go to card three
```

or, instead of writing

```
set the visible of background button 10 to true
```

you can write

```
set the visible of background button ten to true
```

Ordinal Constants

In addition to the numeric constants, there is a series of ordinal constants that can be used to refer to the order of objects. These constants include:

```
first                fifth                ninth
second               sixth                tenth
third                seventh
fourth               eighth
```

Thus, you can write

```
set the visible of the first card field of the sixth card¬
   to false
set the style of the second background button of the first¬
   card to "shadow"
```

Using these constants and ordinal constants is convenient, especially when debugging using the Message box. However, you need to be careful that you don't use them to write scripts that no longer work if you change your stack. For example, if you put this script on a card button—

```
on mouseUp
   set the visible of the third card field to false
end mouseUp
```

and then later delete the second field on the card, the script will no longer work correctly.

Special Ordinals

In addition to the ordinal constants, there are three special ordinals whose values depend on the number of objects in the group to which you are referring. These three ordinals are

```
last
any
mid[dle]
```

As you would expect, **last** refers to the last object in the set, so

```
go to last card
```

puts you on the last card in your stack. If you have seven buttons on your card, then

```
set the textStyle of the last card button to "Bold"
```

changes the text style on the seventh button to Bold.

Middle refers to half the number of objects plus one. Again, if you have seven buttons, typing

```
set the textStyle of the middle card button to "Bold"
```

will set the text style on the *fourth* button. That is, HyperTalk computes

```
7 / 2 = 3.5
```

Using just the whole number, it then computes

```
3 + 1 = 4
```

If you have eight card buttons, the middle button is number 5:

```
8 / 2 = 4 + 1 = 5.
```

Finally, **any** refers to a random item. If you have seven buttons and you type

```
set the hilite of any card button to true
```

a random button among all seven of your buttons will highlight.

Using The

Almost from the beginning of your work with *HyperCard* and HyperTalk you have encountered the use of the word **the**. Often you found that the use of **the** was optional. In general, **the** is used to make HyperTalk code more readable, more English-like. However, there are some guidelines for its use.

You have seen that **the** is required in some functions. That is, it is required before functions that don't have an input *when you don't follow the function with parentheses*. For example, **the** is required in this case—

```
the date
```

but not in this one—

```
date()
```

In all other cases, **the** is optional. It may appear before

- **message** or **selection** (see Chapter 19) or **target** (see your *HyperCard* documentation)
- the name of any property, e.g., **the name of ...**
- functions using **of**, e.g., **the number of ...**

Selection is discussed in Chapter 19 and **target** is discussed in Chapter 27. You will likely want to come back to this somewhat technical description in the future when you have questions about the use of **the**.

Debugging and Testing

In a number of places in this chapter, you are cautioned about using special words in your scripts. The danger in many cases is that you will change your stack in some way and thus change the meaning of your scripts. A command as simple as

```
go to next card
```

no longer works correctly if you insert a new card between the current card and the card that follows it. Most often it is best to use object names when writing scripts. However, using these constants and ordinals can be quite useful when you are debugging by using the Message box. It is often easier to type

```
set the visible of the first button to true
```

rather than to type the full name of the button. On the other hand, using **me** and **this** can sometimes allow you to save writing many different but very similar scripts.

The key to the use of special words is to think carefully about the long-term use of any scripts you write. Is the script you are writing to be used just to test a new idea? Then using specific object names may not be important. Is the script you are writing going to become part of a larger stack? Then it is important to make the script as free of dependency on the current structure of the stack as possible.

New HyperTalk Words in This Chapter

Constants
zero
one
two
three
four
five
six
seven
eight
nine
ten

Ordinal Constants
first
second
third
fourth
fifth
sixth
seventh
eighth
ninth
tenth

Special Ordinals
any
last
mid[dle]

Object Descriptors
me
next
previous
recent
this

Functions
mouseH
mouseV

Project Ideas

Experiment with the special object descriptors. Can you correctly predict the outcome of handlers in which you use **this** and **me**?

Create a card with a number of similar objects on it. For example, create a card with seven or eight buttons on it. Then experiment with the constants, ordinal constants, and special ordinals. Do you understand what is happening as you work with each of them?

Work with the example in this chapter that involved using **me** with the Catch buttons. Experiment with the use of **any**. Perhaps you can turn this into a game. You

might highlight a random button and then ask the user to try to "catch" it. Use this example to experiment some more with the new words in this chapter.

Chapter 13. Working With the Tool Window

In this chapter you will learn how to use HyperTalk to manipulate the Tool window. You will learn that all the icons in the Tool window have names you can use to select them. You will also learn how to use HyperTalk to simulate clicking and dragging with the mouse.

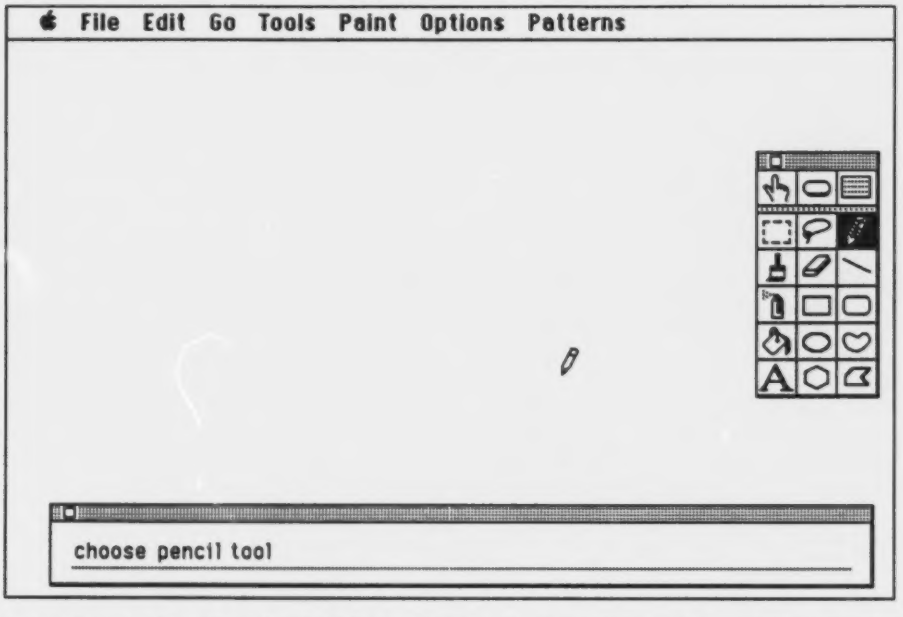
You have learned that you can use HyperTalk to accomplish many kinds of tasks. You have learned to change the properties of buttons and fields, create special effects, play sounds, and put text into fields. In much of your work with *HyperCard* you have made use of the Tool window. This window is essential to much of your work. It should be no surprise to you that you can use HyperTalk commands and functions to create scripts that make use of the Tool window.

Using Tools

Begin with a blank card, tear off the Tool window, display the Message box, and type

```
choose pencil tool
```

Note that you may use **the** in *HyperCard* 2.0, but not in earlier versions of *HyperCard*. You see



Notice that the Pencil tool is highlighted in the Tool window and that the pointer has changed to the pencil.

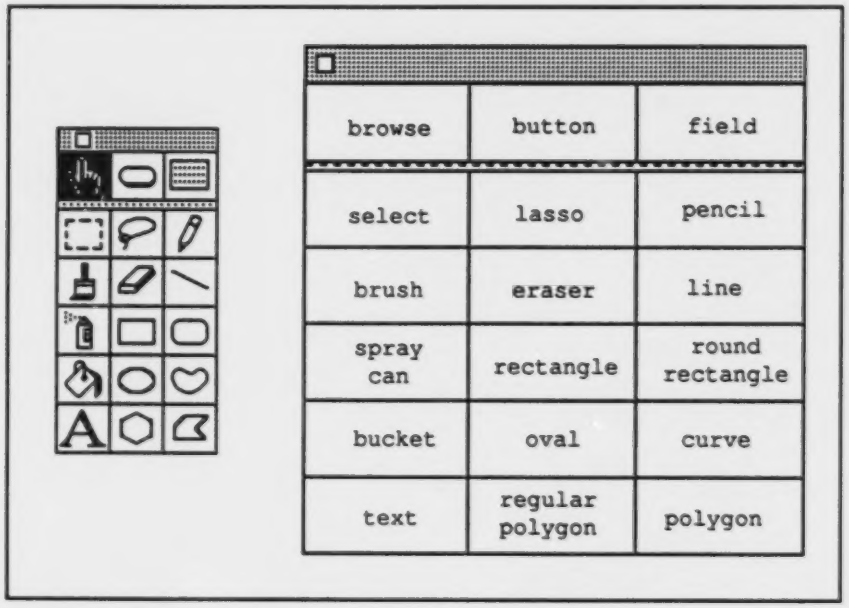
Choose has the syntax

```
choose toolName tool
choose [the] toolName tool  (in HyperCard 2.0)
```

where "toolName" can be any of the following:

`browse`	`field`	`rect[angle]`
`brush`	`lasso`	`reg[ular] poly[gon]`
`bucket`	`line`	`round rect[angle]`
`button`	`oval`	`select`
`curve`	`pencil`	`spray [can]`
`eraser`	`poly[gon]`	`text`

The following diagram shows which names correspond to which symbols in the Tool window:



Experiment with the **choose** command. Note what happens on the screen as you type different **choose** commands in the Message box.

Clicking and Dragging

When you work with the Tool window, you often use the mouse. For example, to draw a rectangle, you choose the Rectangle tool, click on the card, and drag to where you want the other corner of the rectangle. There are two HyperTalk commands that allow you to accomplish this clicking and dragging from scripts. These commands are, not surprisingly, **click** and **drag**. The syntax of **click** is:

```
click at integer1, integer2 [with key1 [,key2[,key3]]]
```

"Integer1,integer2" represent the coordinates of a point on the card. "Key1", "key2", and "key3" are one or more of the constants **shiftKey**, **optionKey**, or **commandKey**. These three constants are used to have the effect of pressing the Shift, Option, or Command keys, respectively. **Drag** has the syntax

```
drag from integer1, integer2 to integer3, integer4
   [with key [,key2[,key3]]]
```

Here "integer1, integer2" and "integer3, integer4" are coordinate pairs separated by commas. The last part of each of these statements is used to add keys that you hold down to modify the effect of a particular tool. For example, if you choose the Select tool and hold down the Command key while you outline a graphics object on the screen, the rectangular "marquee" will snap tightly around the object when you release the mouse button. This same result can be achieved by typing the following command in the Message box:

```
drag from 100,50 to 200,150 with commandKey
```

Once this object is selected, you can then use **drag** again to move the object elsewhere on the screen. Any combination of keys you use when working directly with the Tool window can be duplicated using this last part of the **click** and **drag** commands.

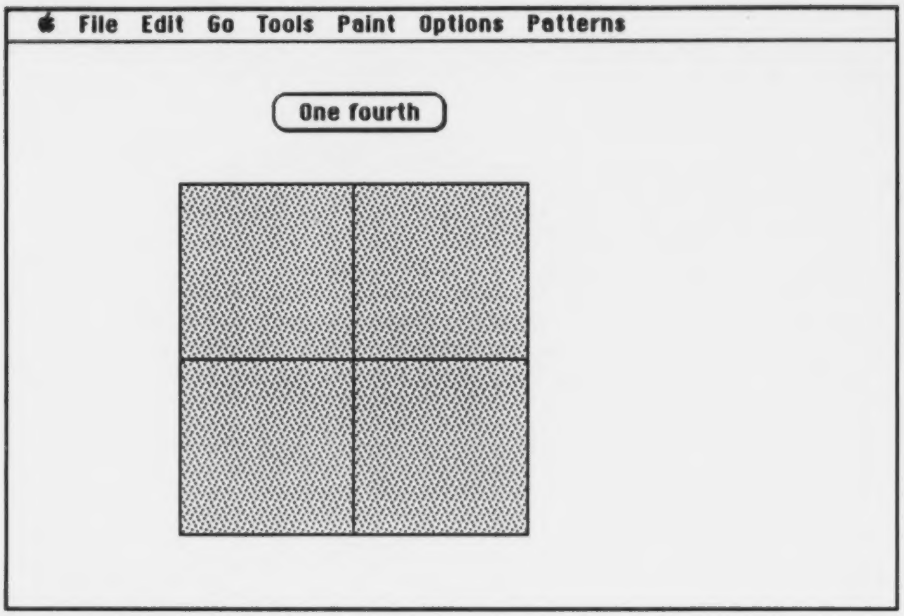
To begin exploring these new commands, create a new card. Using the Message box, enter these commands one at a time to create a square divided into four parts:

```
choose rectangle tool
drag from 100,100 to 300,300
```

Fill the rectangle with a pattern using the Bucket tool. Then divide the square into four parts with

```
choose rectangle tool
drag from 100,100 to 200,200
drag from 200,200 to 300,300
```

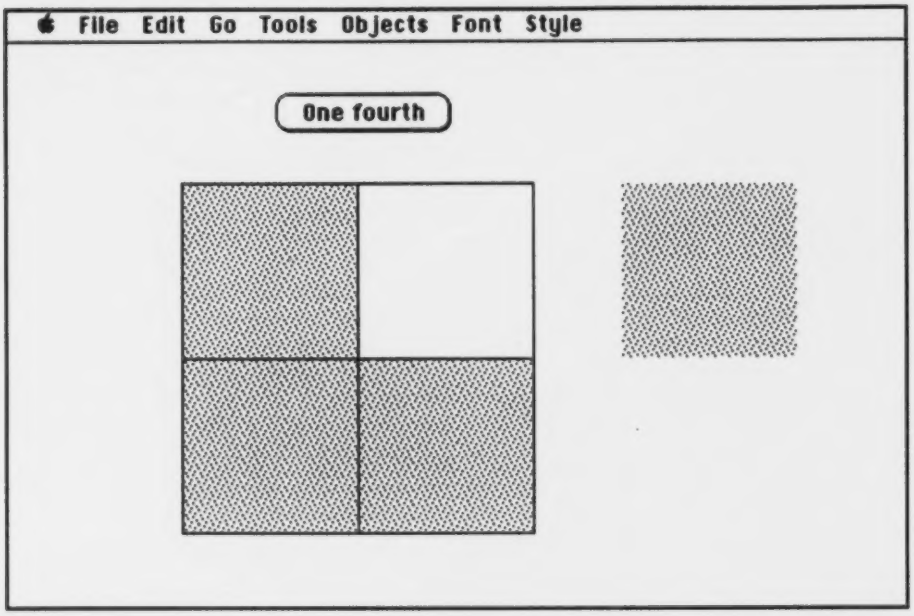
Then add a button called "One fourth":



On the button One fourth, put the following script:

```
on mouseUp
   choose select tool
   --select the interior of one fourth of the square
   drag from 201,101 to 299,199
   --move one fourth of the square to the right
   drag from 250,150 to 400,150
   choose browse tool
end mouseUp
```

Click on the One fourth button. You see one-fourth of the square move to the right:



Notice that the first **drag** command was used to select the part of the object to be moved. That is, the command **drag** includes the mouse click to place the mouse pointer. The second **drag** command is the same as clicking in a selected object and dragging with the mouse. This can be confusing at first. Sometimes it is best to first move the graphic "by hand" to see what you do with the mouse. Then transfer those movements into **click** and/or **drag** commands.

Notice also that the handler ends with choosing the Browse tool. It is generally a good idea to end handlers this way so that *HyperCard* is back to a "normal mode" in order for the user to be able to click on objects in the usual manner.

When you click on the One fourth button, the shaded square may not move to the right at a speed that you like. You can change the speed of this movement using the global property **dragSpeed**. (See Chapter 8 for a discussion of global properties.) The input to **dragSpeed** determines how many pixels (dots) per second the pointer moves across the screen. There are 72 pixels per inch on the standard Macintosh screen. **DragSpeed** is automatically set to 0 when *HyperCard* is idle. Thus, if you modify the above script, adding **dragSpeed** as follows,

```
on mouseUp
   choose select tool
   --select the interior of one fourth of the square
   drag from 201,101 to 299,199
   set the dragSpeed to 72
   --move one fourth of the square to the right
   drag from 250,150 to 400,150
   choose browse tool
end mouseUp
```

the square moves slowly to the right. If you make the number after **dragSpeed** bigger, then the square moves faster.

When working with a script such as this one, it can be a nuisance to put the card manually back to normal each time you click on the button. One strategy to make this easier is to put a comment mark (--) in front of the

```
choose browse tool
```

command in the script. Then, when the square has finished moving, you can use **Undo** from the **Edit** menu (or press Command-Z). When the script is working exactly as you wish, remove the comment mark. Another technique is to create a script to restore the graphics. For example, you might write

```
on putBack
   choose select tool
   --select the separate one-fourth
   drag from 350,100 to 450,200
   --move the one-fourth back
   drag from 400,150 to 250,150
   choose browse tool
end putBack
```

You can then put this script on a temporary button or on the One fourth button, and then type

```
send "putBack" to card button "One fourth"
```

Using one of these techniques can save a lot of debugging time.

Painting Properties

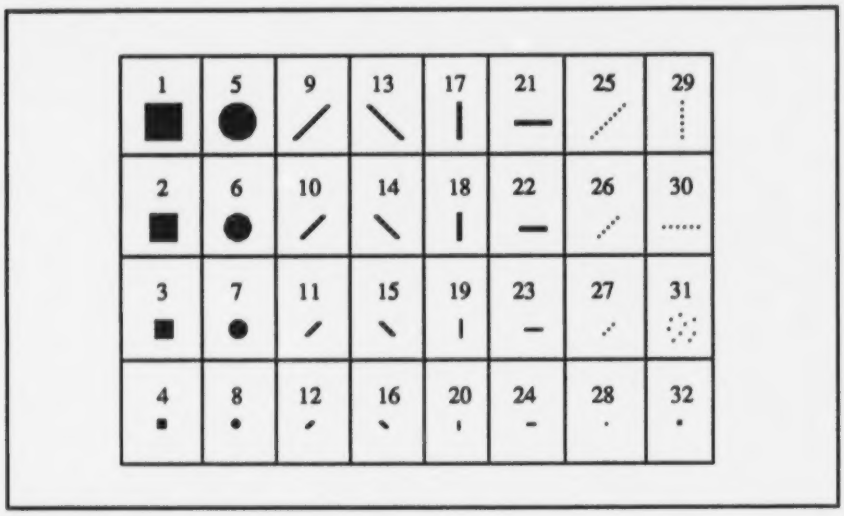
You have learned about the properties of buttons and fields, about global properties, and about window properties. There are also painting properties. These painting properties are found on a number of the menus you use with graphics. For example, some of the properties are listed under the **Options** and **Patterns** menus. Painting properties are also available from **Text Style** on the **Edit** menu.

The properties you can select from the **Options** menu,

include

`brush`	`grid`	`multiSpace`
`centered`	`lineSize`	`polySides`
`filled`	`multiple`	

The **brush** property can be set to any of 32 different brush styles shown in the following chart:



Thus, to select a large square brush, you type

```
set brush to 1
```

The **centered, filled,** and **multiple** properties can have the values **true** or **false**. All three of these properties are set by default to **false**. Setting them to **true** with a statement such as

```
set centered to true
```

is the equivalent of checking **Draw Centered** under the **Options** menu. Recall that **Draw Centered** causes shapes such as circle and squares to draw from the center rather than from the edge.

Similarly, typing

```
set filled to true
```

is the equivalent of checking **Draw Filled**, while typing

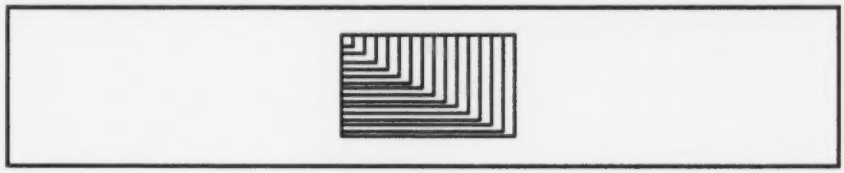
```
set multiple to true
```

is the equivalent of checking **Draw Multiple**.

Closely associated with the **multiple** property is the **multiSpace** property. This determines the space between multiple images drawn by shape tools when **multiple** is **true**. **MultiSpace** can be any number from 1 to 9; by default it is set to 1. Thus, if you type

```
set multiSpace to 5
```

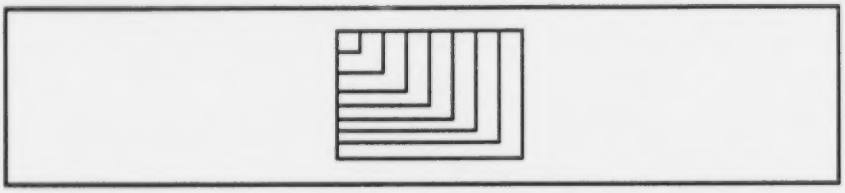
in the Message box, then choose the rectangle tool, and click and drag slowly, you see



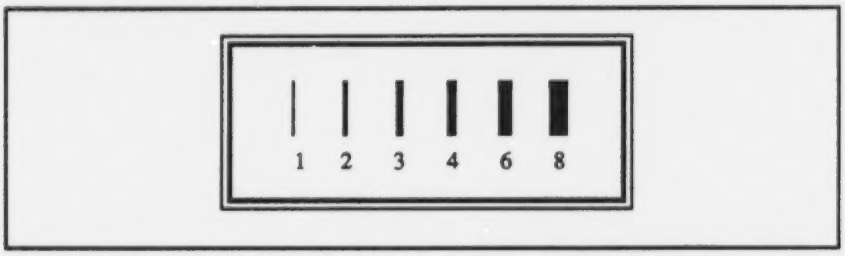
but if you type

```
set multiSpace to 10
```

and click and drag slowly, you see

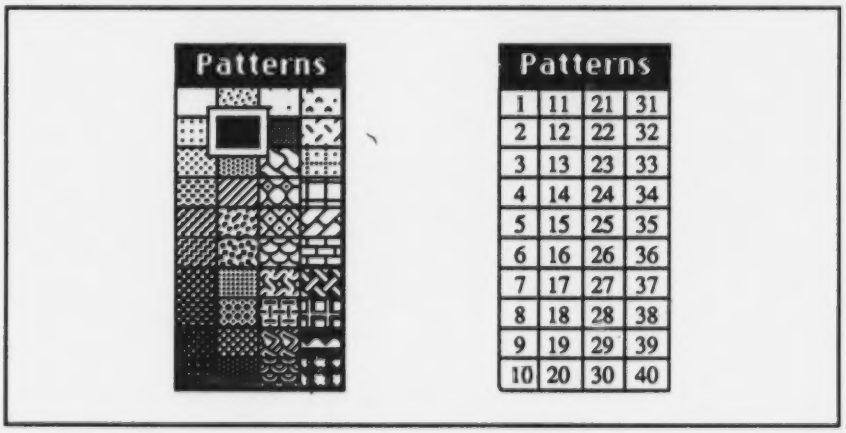


The **lineSize** property determines the thickness of the line drawn by the Line and Shape tools. It can have the values 1, 2, 3, 4, 6, or 8, but is automatically set to 1.



PolySides is used to determine the number of sides when the Regular Polygon tool is selected. **PolySides** can have values from 3 to 50. The **grid** property is used to turn the grid on and off. By default, **grid** is set to **false**. When set to **true**, many paint tools are constrained to moving at eight-pixel intervals, just under one-eighth of an inch.

Patterns can be selected from the Pattern window. In the following illustration, the chart to the right shows the numbers corresponding to various patterns. (Note that each stack has its own Pattern window, so if you change the patterns on one stack they will not be the same on the next stack you open.)



Thus, you can change the current pattern being used by typing

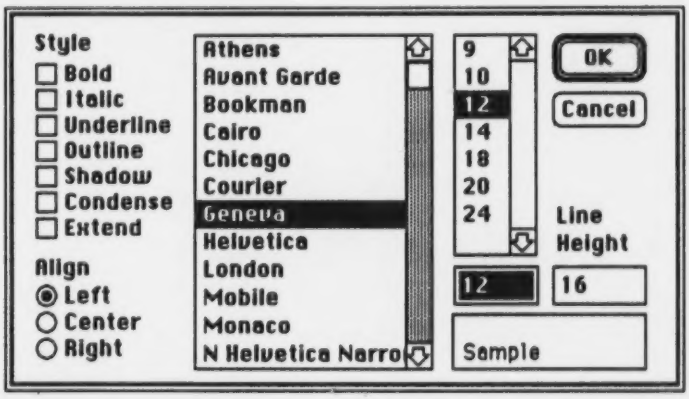
```
set pattern to 36
```

The pattern is then set to the horizontal brick pattern.

Finally, just as you can change the text style on buttons and in fields, so too you can change the style of text that you can "paint" on the card or background. These painting properties,

```
textAlign
textFont
textSize
textStyle
textHeight
```

correspond to the following dialog box:



Thus, a handler to create paint text the same as the paint text chosen in this dialog box would be

```
on mouseUp
   choose text tool
   set the textAlign to "left"
   set the textFont to "Geneva"
   set the textSize to 12
   set the textStyle to "Plain"
   set the textHeight to 16
end mouseUp
```

You can work with paint text properties in exactly the same way that you worked with text properties in fields and on buttons.

Finally, you can restore all the paint properties to their original settings by using the **reset paint** command. The syntax for the **reset paint** command is:

```
reset paint
```

Using this command resets the paint properties.

Property	Reset paint sets value to:
brush	1
centered	false
filled	false
grid	false
lineSize	1
multiple	false
multiSpace	1
pattern	12 (black)
polySides	4
textAlign	left
textFont	Geneva
textHeight	textSize + one-third of textSize
textSize	12 and textStyle to plain.

Debugging and Testing

Recall that you can use the function **mouseLoc** to determine screen coordinates. You place the mouse pointer at a point on the screen and then type

```
the mouseLoc
```

in the Message box. Coordinates for the location of the mouse pointer appear. You can then use these coordinates in **click** and **drag** commands.

Don't forget that you can use **Undo** from the **Edit** menu—or press the Tilda key (~/`)—to restore graphics to their original state after a single change. Remember that you must use **Undo** immediately after you make a change.

Be sure to remember to put

```
choose browse tool
```

at the end of scripts that use the Tool window so that the user can continue to work with your stack.

As you have learned by now, *HyperCard* automatically saves as you work with your stack. However, the automatic saving works differently when you are using the Paint tools. Graphics are *not* saved unless you

- stop using the Paint tools
- go to a different card
- use **Keep** (Command-K) from the **Paint** menu

If you are unhappy with your graphics changes, you can use **Revert** from the **Paint** menu to get back the previously saved graphics. In Chapter 16 you will learn how to use the **doMenu** command to use **Keep** and **Revert** from HyperTalk scripts.

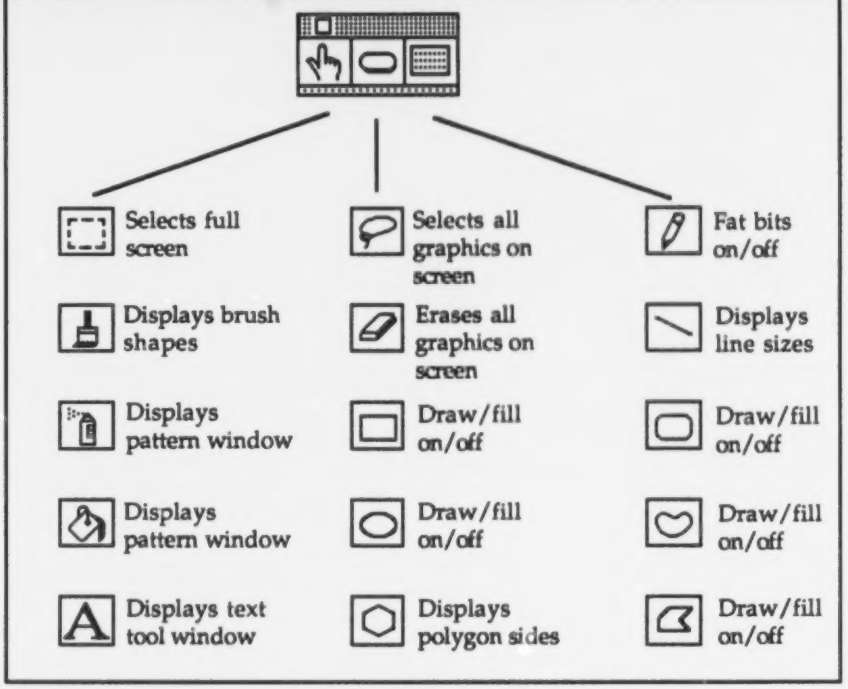
Recall that there is an important difference between "paint text"—text created using the Text tool—and text that is put into fields using the I-beam pointer. Text in fields can be edited in much the same manner that you edit text in a word processor. However, once you move the mouse pointer to a new location or leave the Text tool, you can no longer edit paint text at all. You must erase it just like any other graphic.

Keyboard Shortcuts

Menu Bar Commands	
Command-S	select graphics on the screen
Command-A	select all
Command-K	keep
Command-T	text style

Double Click On Tool	
Select	selects the full screen
Lasso	selects all the graphics on the screen
Pencil	sets fatbits mode on/off
Brush	brings up brush shapes window
Eraser	clears all graphics on the screen
Line	brings up line sizes window
Rectangle	sets draw filled mode on/off
Round rectangle	sets draw filled mode on/off
Curve	sets draw filled mode on/off
Oval	sets draw filled mode on/off
Polygon	sets draw filled mode on/off (doesn't draw filled)
Bucket	show/hide patterns window
Text	bring up text styles window
Regular polygon	bring up polygon sides window

Double-click on the Paint Tools



Shift and Drag On/With:	
Selected graphic	moves the graphic horizontally/vertically
Pencil tool mode	limits to horizontal or vertical lines
Brush tool	limits to horizontal or vertical lines
Eraser tool	limits to erase horizontally or vertically
Line tool	constrains lines to 15 degree intervals
Spray can	limits to spraying horizontally or vertically

Command, Click and Drag With:	
Select tool	selects exactly a rectangle surrounding a graphic
Pencil tool	fatbits mode on/off
Lasso tool	select all
Brush tool	brush in white pattern
Spray can	spray in white pattern

Option, Click and Drag On/With:	
Selected graphic	copy that graphic
Select tool	same as lasso tool

Shift, Command Click and Drag With:	
Spray can	spray horizontally/vertically in white pattern

Shift, Option Click and Drag On:	
Selected graphic	copy and move horizontally/vertically

Option, Command Click and Drag With:	
Pencil tool, fatbits mode	move the window around

Delete Key With	
Selected graphic	removes the graphic

New HyperTalk Words in This Chapter

Commands
`choose`
`click`
`drag`

Global Properties
`dragSpeed`

Painting Properties
brush
centered
filled
grid
lineSize
multiple
multiSpace
polySides

Paint Text Properties
textAlign
textFont
textSize
textStyle
textHeight

Project Ideas

Now that you can control all of the painting capabilities from HyperTalk, try creating an entire picture or design from scripts. This is an excellent opportunity to practice writing your own handlers and writing modular scripts.

Practice using **Keep**, **Revert**, and **Undo**. All of these menu commands are useful when you are working with graphics. Using them thoughtfully can save you a lot of debugging time.

Did you note that you can select both the Button and Field tool from the Tool window using **choose**? Think for a moment about **click** and **drag**. Can you create a button that contains a script to move itself? How about moving another button or a field? How about creating a script that allows the user to watch a "painting" be slowly drawn?

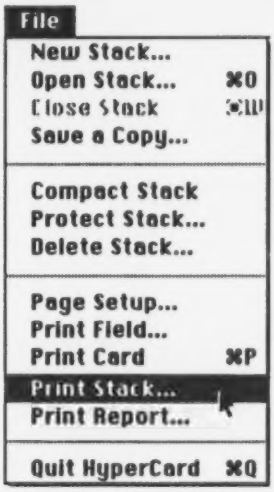
Chapter 14. Printing

This chapter gives a brief overview of printing using the options available under the **File** menu. In addition, it explores the use of the **print** command from scripts.

In general, documents produced using *HyperCard* are designed for use on the computer. It is virtually impossible to reproduce on paper the nonsequential nature of many *HyperCard* stacks. However, there are times when you want to print part or all of your stack. While the strength of *HyperCard* is not in its printing capabilities, it does provide a variety of printing options.

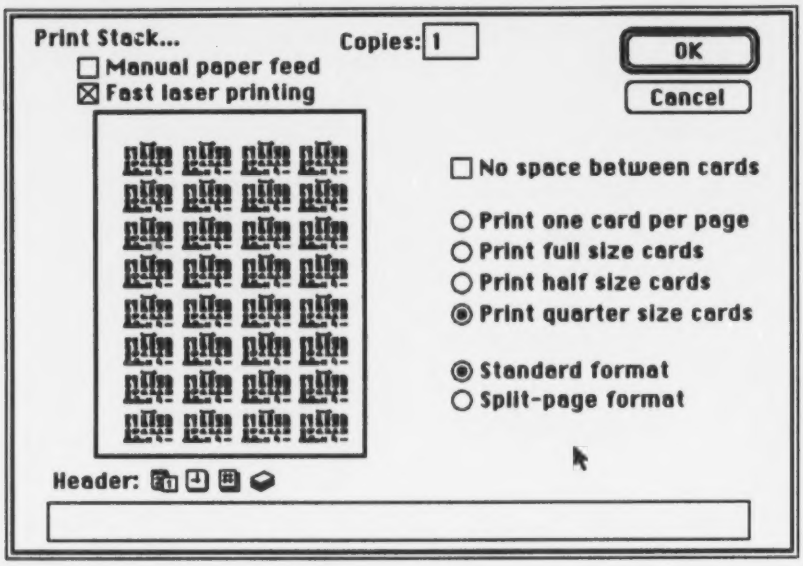
Printing Stacks

If you want to print your entire stack, you can select **Print Stack** from the **File** menu. (Note that you must have the Browse tool selected to access the print choices on this menu.)

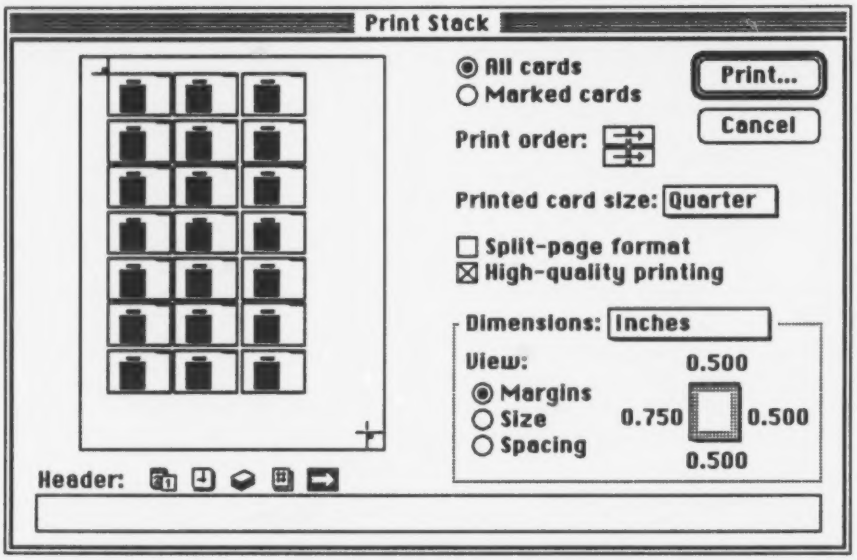


If you are using a version of *HyperCard* before 2.0, this menu will look slightly different.

In a version of *HyperCard* earlier than 2.0, you then see something like

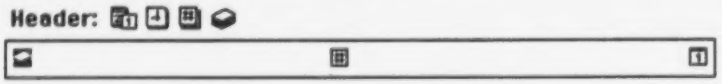


If you are using *HyperCard* 2.0, you see

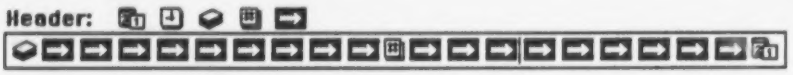


This dialog box provides a wide variety of options for printing your stack. You should see the documentation for your particular version of *HyperCard* to learn about all of the options. There are several options that are similar among versions of *HyperCard*. For example, printing *HyperCard* stacks can take a long time. If you want the printing to be faster (and of lower quality), you check the **Fast laser printing** box in earlier versions of *HyperCard* and you *uncheck* the **High-quality printing** box in *HyperCard* 2.0. Note also that in both versions of the dialog box you have the option of determining how big the cards will be on the page. In *HyperCard* 2.0, you click in the box after **Printed card size:** and a menu of possible sizes appears.

At the bottom of both of the **Print Stack** dialog boxes is the word **Header** followed by a number of small icons. The empty rectangle below these icons allows you to design the heading that will appear at the top of each page when you print your stack. The first icon represents the date. The second represents the time. Next comes the page number, and finally the stack name. When you click on an icon, it appears in the box. If you are using a version of *HyperCard* before 2.0, you can move the icons around using the mouse and the space bar. The arrangement of icons shown in the following illustration causes each printed page to have the stack name in the upper left corner, the page number in the center, and the date in the upper right corner.

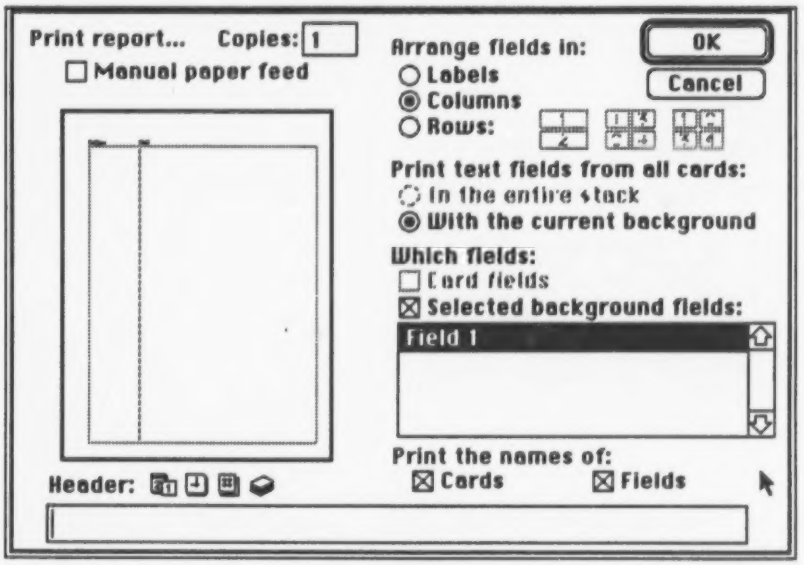


If you are using *HyperCard* 2.0, there is an additional "tab" icon. A *HyperCard* 2.0 header to put the stack name on the left, the page number in the center, and the date in the right corner would look like this:

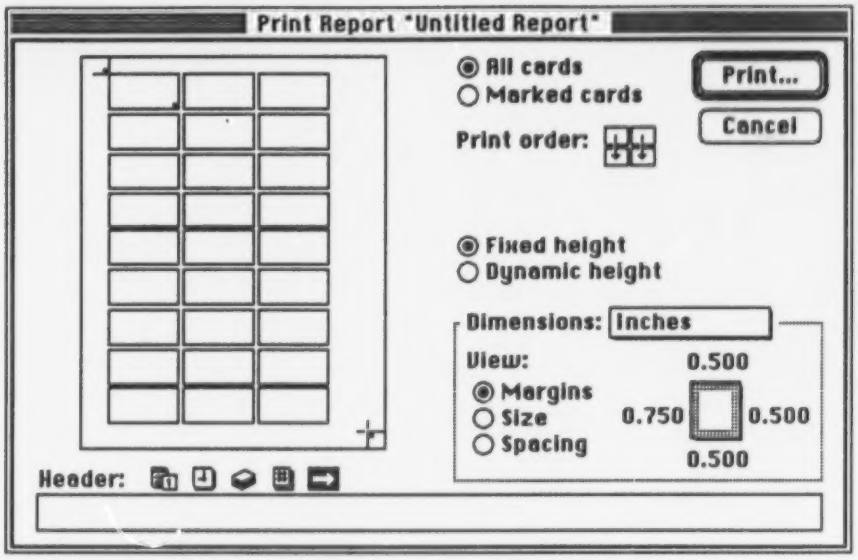


Printing Reports

There is also a **Print Report** option in the **File** menu. If you select this option in a version of *HyperCard* before 2.0, you see something like



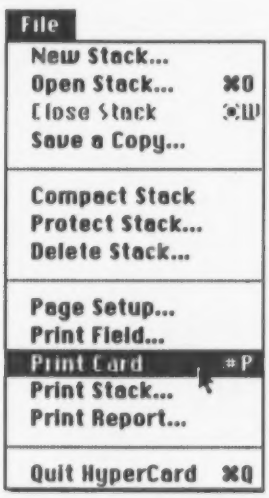
In *HyperCard* 2.0 you see



These dialog boxes provide a wide range of options for displaying the text contained in fields in your stack. You should look at the documentation for your version of *HyperCard* for the details on how this dialog box works.

Printing Cards

If you select **Print Card** from the **File** menu, the card currently displayed on the screen is printed.



Note that you can use the Command-P shortcut to print a single card.

In addition to using dialog boxes to print cards, you can also use the **print card** command. This command can be used from the Message box or from a script. The syntax of the **print card** command is:

```
print card
print expression card[s]
print cardDescriptor
```

The first form, **print card**, causes only the current card to be printed. This form of the command works exactly like the **Print Card** option in the **File** menu. The second form of the **print card** command allows you to specify which cards you want printed. Here are some examples:

```
print first card
```

to print the first card in the stack,

```
print all cards
```

to print all of the cards in the stack,

```
print next card
```

to print the next card in the stack beyond your current location in the stack, or

```
print 5 cards
```

to print five cards from your stack beginning with the card you are on.

The third form of the **print** command allows you to specify a particular card to print. From any card on your stack, you can type

```
print card id 2013
```

or

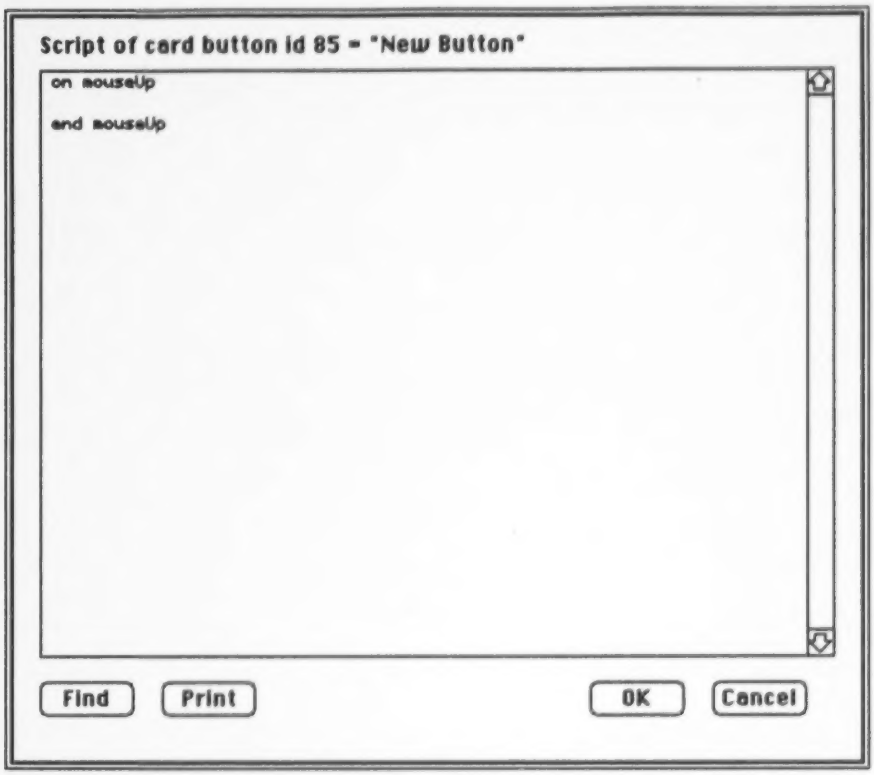
```
print card "Demo"
```

to print a different, particular card in the stack. This form of the command causes *HyperCard* to go to the named card, print it, and return to the current card.

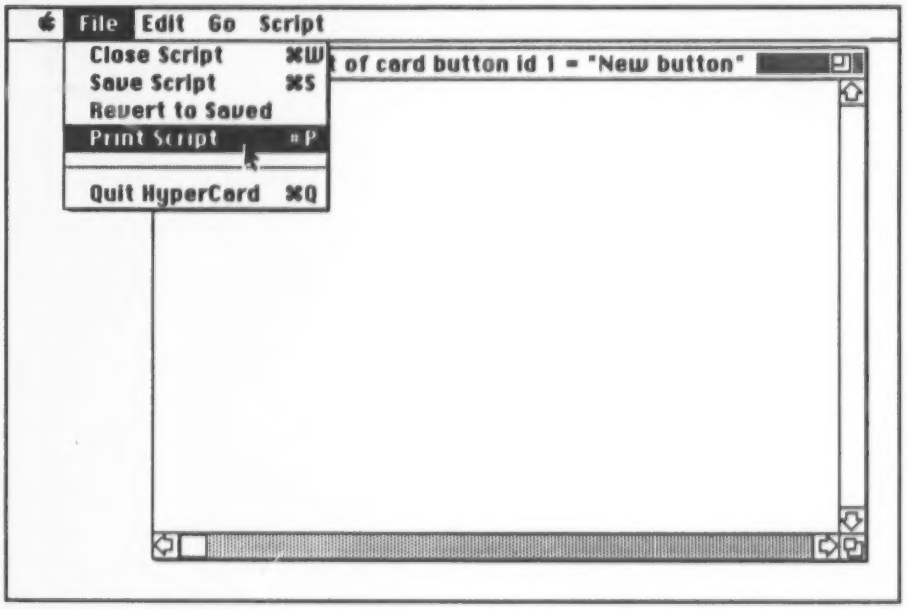
When you use the second and third form of the **print card** command, the cards are printed in the size currently indicated in the **Print Stack** dialog box. You can change the form used by clicking the appropriate boxes in that dialog box and then clicking on the **Cancel** button. The next time you use the **print card** command, the cards will be printed as you specified in the dialog box.

Printing Scripts

When you are writing scripts in versions of *HyperCard* before version 2.0, there is a Print button at the bottom of the Script editor:



Clicking on the button will print the entire script in that particular script editor. If you are using *HyperCard* version 2.0, you must select the **Print Script** option from the **File** menu *when the script editor is on the screen*:



Sometimes you want to print only a part of your script. If you highlight the part of the script you want to print, only that portion will be printed when you click on the **Print** button or select **Print Script**.

New HyperTalk Words in This Chapter

Commands
`print card`
`print` *`expression`* `cards`
`print` *`cardDescriptor`*

Key Shortcuts	
Command-P	print a card

Project Ideas

Spend some time experimenting with the various forms of the print commands. Study the dialog boxes. Examine the documentation that came with your version of *HyperCard*. Do you understand what each part of the box does? Try different report arrangements. Can you modify these different reports to your satisfaction?

Chapter 15. Animation

In this chapter you will explore two different ways to do animation. You will make use of techniques you already know and, in addition, learn a simple form of the **repeat** structure.

You have already created some simple movement in your work with HyperTalk. For example, in Chapter 12, you created buttons that moved when you clicked on them:

```
on mouseUp
   set the hilite of me to true
   wait 1 second
   set the visible of me to false
   set the location of me to the mouseH + 20,¬
      the mouseV + 20
   wait 1 second
   set the hilite of me to false
   set the visible of me to true
end mouseUp
```

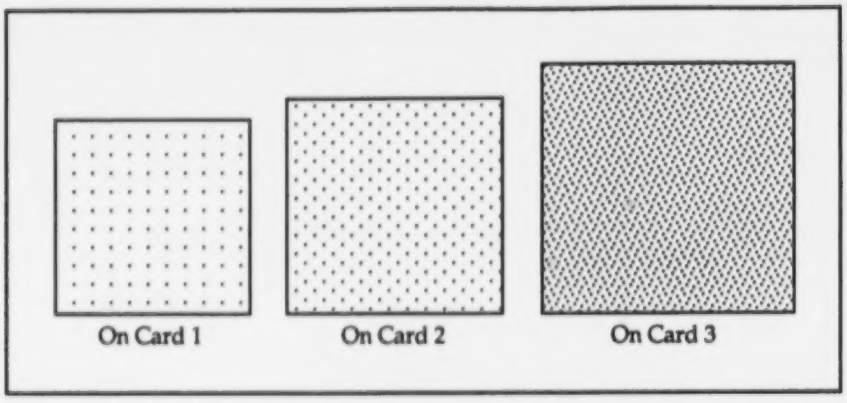
In Chapter 13 you also moved part of a graphic by using **drag**:

```
on mouseUp
   choose select tool
   --select the interior of one fourth of the square
   drag from 201,101 to 299,199
   --move one fourth of the square to the right
   drag from 250,150 to 400,150
   choose browse tool
end mouseUp
```

These simple examples give a hint of some ways you can create animation in *HyperCard*.

Flipping Cards

One simple way to create animation is to create a series of cards, each with a slightly different picture on it. Then you can flip through the cards to produce an animation effect, much like you might flip through a real deck of cards or flip book on which each card or page has a slightly different picture. Begin by creating a card with a filled square on it. Then make a slightly darker and larger square on the next card and a yet darker and larger square on the third card:



If you want the square to be placed accurately, you might want to use a combination of selection tools from the Tool window and HyperTalk commands. Select the pattern you want, and then choose the **Draw Filled** choice from the **Options** menu. With the Rectangle tool selected, type

```
drag from 100,100 to 200,200
```

in the Message box. Select a different pattern for each rectangle and use HyperTalk to place them:

```
drag from 90,90 to 210,210
drag from 80,80 to 220,220
```

If you now "flip through" the cards, you will see the square get larger and darker. You can flip through the cards by using Command-3 to go quickly to the next card. If you are on the first card, you can also use

```
show 2 cards
```

to see the next two cards, but on some newer Macintosh computers this may occur so quickly that you can't really see the animation.

When you are satisfied with your animation, add an "Animate" button to your card. There is a fast shortcut for creating buttons. Choose the Button tool, then hold down the Command key. Click and drag, and a transparent button appears. (The same technique works for fields.) Make the button any style you want and then add a script to flip through the cards. You might write

```
on mouseUp
   wait 15 ticks
   go to next card
   wait 15 ticks
   go to next card
end mouseUp
```

You will need to adjust the timing for your particular machine and preference.

This particular animation technique works fine for a few cards. However, if you have a lot of cards to animate, your handler can get quite long. There is a more efficient way to repeat actions a number of times. You can use the **repeat** keyword. The syntax of the **repeat** structure is

```
repeat controlForm
   statementList
end repeat
```

There are a number of ways to use the **repeat** structure. You will learn a number of "control forms" for **repeat** in Chapter 24. For simple animation, you can use the **repeat for** form of **repeat**:

```
repeat [for] number [times]
   statementList
end repeat
```

Thus, the script on the Animate button could be written

```
on mouseUp
   repeat 2 times
      wait 15 ticks
      go to next card
   end repeat
end mouseUp
```

Generally this is an easy and efficient way to create animation using a number of cards.

Of course, **repeat** can be used in other ways than just for animation. Earlier you saw that you could use the special ordinal **any** to select a random object. Thus, you might write a script that would cause a number of different buttons to be highlighted using **repeat**:

```
repeat for 5 times
   set the hilite of any card button to true
end repeat
```

Up to five randomly selected buttons will be highlighted. (It is possible that **any** will select the same button more than once.)

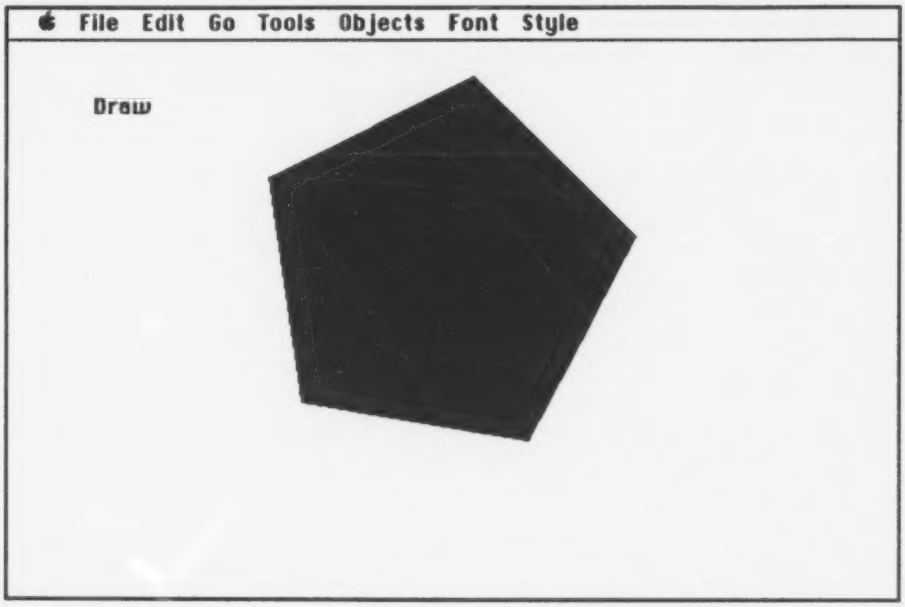
Using the Tool Window

In Chapter 13 you saw that you could move graphics using HyperTalk commands to access the tools on the Tool window. You also saw that you could use **dragSpeed** to control the speed of movement. There are two ways you can create animation effects using these techniques.

The first technique involves setting the **dragSpeed** and then drawing an object slowly. The second technique involves selecting an object and moving it with **drag**, much as was done in Chapter 13. To explore this first technique, create a card with a Draw button in the upper left corner and put this script on it:

```
on mouseUp
   choose regular polygon tool
   set polySides to 5 --draw a pentagon
   set centered to true --draw from the center
   set pattern to 20 --pick a dark pattern
   set dragSpeed to 50 --set the speed
   drag from 250,150 to 300,250 --draw the pentagon
   choose browse tool
end mouseUp
```

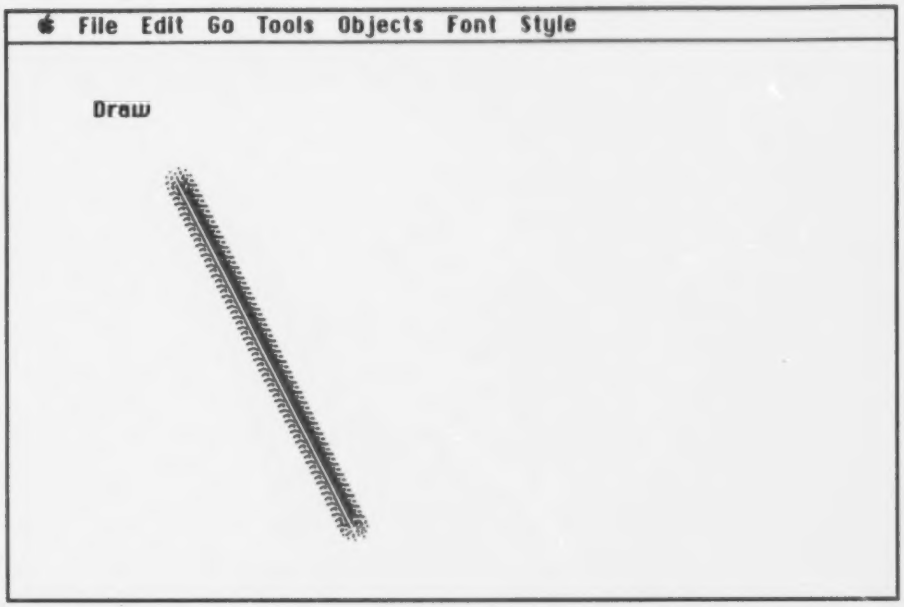
When you click on your Draw button, you see a pentagon slowly grow from a dot at the center:



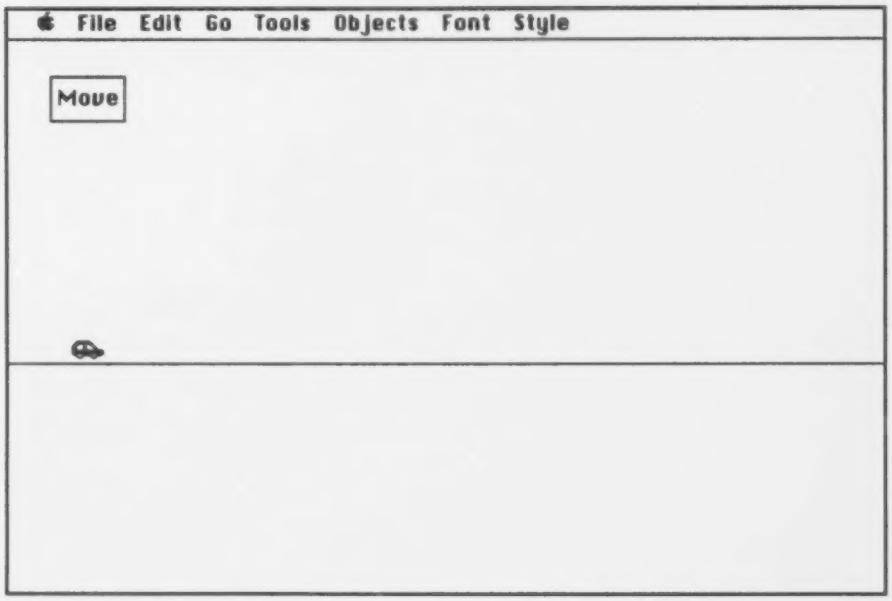
Adjusting the **dragSpeed** in the above handler will change the effect. Try it!

Drawing slowly is an easy way to create shapes or effects that can be achieved using just the tools in the Tool window. For example, you can get interesting effects with the Spray tool:

```
on spray
   set the pattern to 12
   choose spray can tool
   set the dragSpeed to 100
   drag from 100,100 to 200,300
   choose browse tool
end spray
```



The second technique for creating animation using the Tool window involves moving an existing object. Suppose you have collected some pieces of art that you can paste onto your card. For example, suppose you have a little car pasted at approximately 50, 200:



Then you can add the following script to the Move button:

```
on mouseUp
   choose select tool
   --set the speed
   set the dragSpeed to 100
   --select the car
   drag from 35, 190 to 60, 207 with optionKey
   --move the car
   drag from 40, 200 to 400, 200
   choose browse tool
end mouseUp
```

Clicking on the Move button causes the car to drive across the card. Note that you may have to adjust the numbers after each **drag** so that the picture you paste on your card is properly selected.

Notice that the constant **optionKey** is used in the first **drag** command. Making use of the key shortcuts and options described at the end of Chapter 13 provides powerful ways to use the Paint tools from scripts.

Debugging and Testing

When you are working with animation, it is easy to create some art and then accidentally destroy it as you are experimenting. When you have created a card you want to keep, make a copy of it. Then if you make an error, you can copy the original art back onto your working card and try again.

Don't forget about **Revert, Keep,** and **Undo** (see Chapter 13) when you are working with animation projects. Each of these may be useful at one time or another.

When you are working with an animation technique such as the one described for moving the car, it is easy to think that your script is not working when it is actually correct. If you make an error in one of the numbers, you may be selecting the wrong area of the screen or clicking on the wrong spot to "pick up" the graphic you want to animate. Try creating a blank card. Paste a copy of your button with an animate script on it on that new card. Fill the card with a pattern and then try your button. In the case of the Move button shown previously, you see a rectangle of the pattern cut out and moved to the right. Using this technique can give you an idea of whether your handler is working at all.

New HyperTalk Words in This Chapter

Keywords

`repeat [for] number [times]`
`   statementList`
`end repeat`

Keyboard Shortcuts	
Command-drag	create a new button when button tool is selected
Command-drag	create a new field when field tool is selected

Project Ideas

There is a variety of projects that can be developed using animation. Experiment with the different techniques used in this chapter. Which techniques are best for which types of animation? Here are some ideas to try.

- There are many public domain collections of art you can get for use with *HyperCard* (and other graphics programs). Often this is called "clip art." Locate some clip art that interests you and create an animated story. The buttons could be used to cause a variety of actions to occur.
- Create some animated "modern art." You could use a variety of geometric shapes and patterns to create a "painting."
- Create some "art tools" that others could use. That is, you might have a combination of a button and a small piece of art. These could be copied to another card and used to cause some simple animation, like the moving car described in this chapter.

Chapter 16. DoMenu

You have learned that most of the things you can do using the mouse in *HyperCard* can be duplicated by using HyperTalk commands. In this chapter you will learn how to use HyperTalk instead of the mouse to work with pull-down menus.

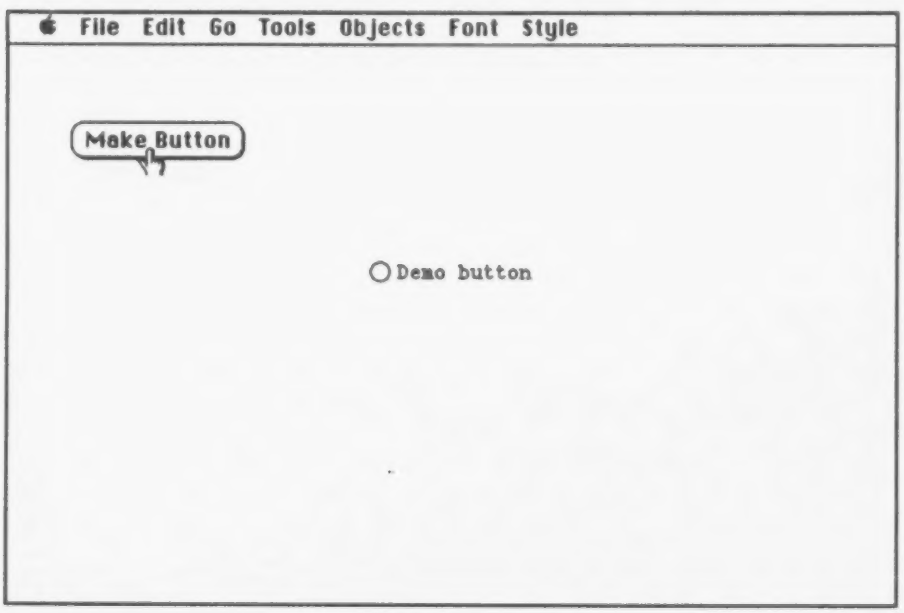
Much of what you do with *HyperCard* involves working with pull-down menus. You have learned that you can do such things as print, set properties, and use Paint tools by using HyperTalk commands. However, there are still many menu options for which you don't have equivalent HyperTalk commands. This problem is solved with the command

```
doMenu menuItem
```

DoMenu allows you to write scripts to do anything that you can do using the mouse and pull-down menus. To explore this idea, create a button called "Make Button" and put the following script on it:

```
on mouseUp
   doMenu "New Button"
   set name of card button "New Button" to "Demo button"
   set the style of card button "Demo button" to radioButton
   set the textFont of card button "Demo button" to "Courier"
   set the autoHilite of card button "Demo button" to true
   choose browse tool
end mouseUp
```

When you click on this button you see



The entire process of creating a new button was done from a handler. You can create any kind of new *HyperCard* object you want by using the **doMenu** command.

DoMenu can be useful in a wide variety of handlers. For example, if you want to allow users to print your stack, you can put

```
doMenu "Print Stack..."
```

into a script. The **Print Stack** dialog box will appear, and users can make the appropriate selections and then print the entire stack without having to access the **File** menu. Note that menu selections such as this one *must* be followed by the three periods that appear when you pull down the menu. When you use **doMenu**, the menu selection you type must be *exactly* the same as what appears when you pull down the menu. In addition, the menu selection must also be available when you use the **doMenu** command. For example,

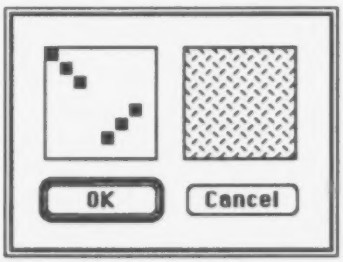
```
doMenu "Print Stack..."
```

will only work correctly if the Browse tool is selected.

You can also use **doMenu** to access menu items available only when you have chosen a Paint tool. Thus, you might include in a script

```
set the pattern to 32
choose select tool
doMenu "Edit pattern..."
```

The pattern-editing window appears so that the user can then change the pattern:



DoMenu can be used effectively to create stacks to work with graphics. To explore such uses of **doMenu**, begin by creating a card with two buttons named "Draw" and "Erase." The Draw button will be used to create random designs on the card; the Erase button will be used to clear the card.

Earlier you learned that the special ordinal **any** can be used to select a random object. There is a function, **random**, that can also be used to create random effects. The syntax of **random** is:

```
the random of integer
random(integer)
```

where "integer" must be positive. Thus, if you type

```
random(40)
```

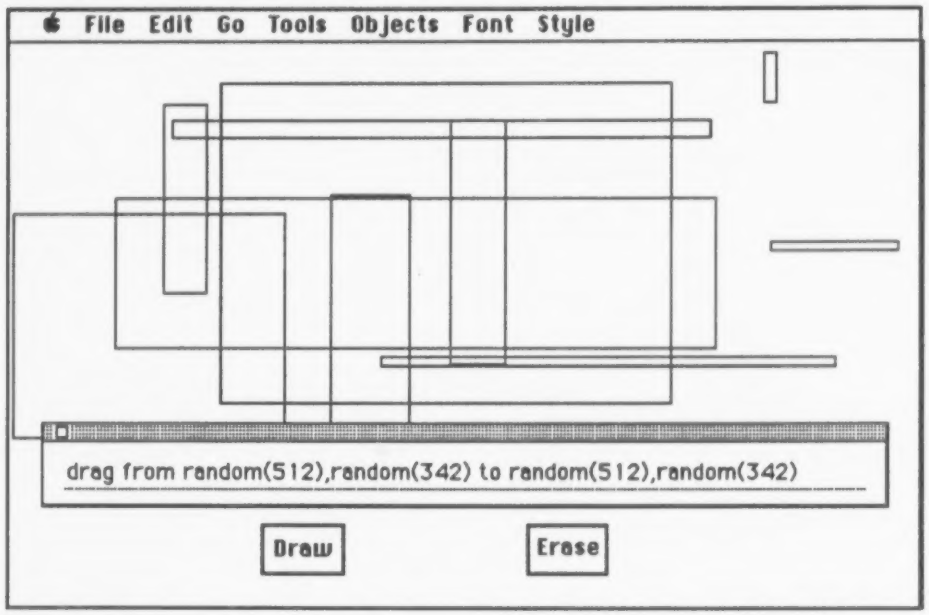
you see a number between 1 and 40 in the Message box. You can use **random** to create a variety of random effects. Tear off the Pattern window and type

```
set pattern to random(40)
```

in the Message box. You see a randomly selected pattern highlight. Click in the Message box and press Return again and you will most likely see a different pattern highlight—although it is possible that the same pattern will be selected several times. Next, select the Rectangle tool from the Tool window. Type

```
drag from random(512),random(342) to random(512),random(342)
```

in the Message box. What happens? Press Return a number of times. Your card should look something like this:



Think about what is happening. The first **random(512)** produces a number between 1 and 512 (the width of the standard *HyperCard* card); the first **random(342)** produces a number between 1 and 342. These two numbers become the top left coordinates of the rectangle. The next pair of random statements produces another set of coordinates. The **drag** command then creates a rectangle with the bottom right corner at the second set of coordinates.

With these new ideas, you can now write a handler that will draw a random rectangle filled with a random pattern:

```
on drawRandomRect
   choose rectangle tool
   set pattern to random(40)
   set filled to true
   --draw a random-sized rectangle
   drag from random(512),random(342) to random(512),¬
      random(342)
end drawRandomRect
```

Put this handler on the Draw button. Recall that you can test it by typing

```
send "drawRandomRect" to card button "Draw"
```

You can then press Return a number of times to see a number of rectangles filled with different patterns.

The **Invert** choice on the **Paint** menu reverses black and white. This handler *selects* a random rectangular area on the card and inverts the pixels in that area:

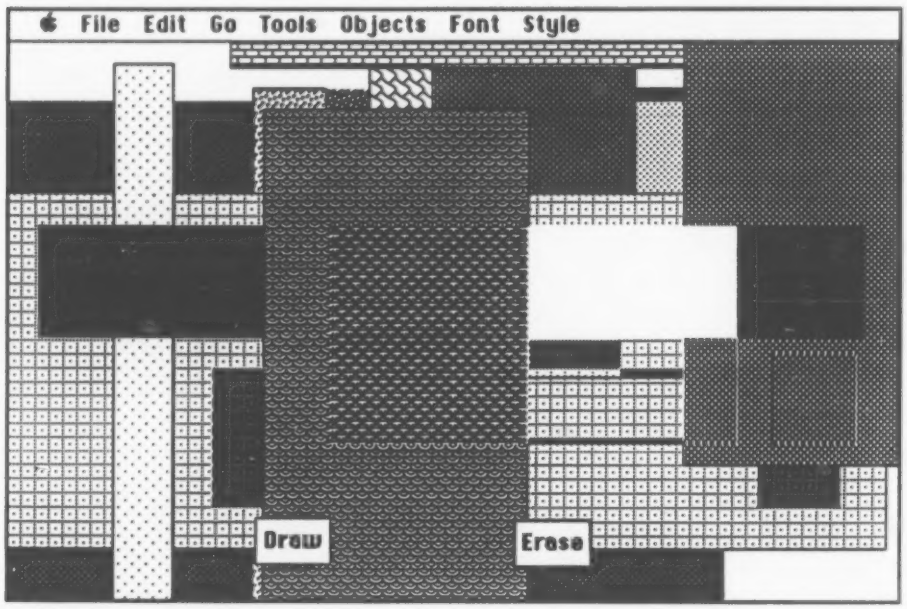
```
on invertRect
   choose select tool
   drag from random(512), random(342) to random(512),¬
      random(342)
   doMenu invert
end invertRect
```

Put this handler on the Draw button and then test it to be sure it works as you expect.

Now, these two handlers can be combined using a **repeat**:

```
on mouseUp
   repeat 10 times
      drawRandomRect
      invertRect
   end repeat
   choose browse tool
end mouseUp
```

Put this handler on the Draw button and test it. You will see something like this:



On the Erase button, put this script:

```
on mouseUp
   choose select tool
   doMenu "Select all"
   doMenu "Clear picture"
   choose browse tool
end mouseUp
```

Clicking on the Erase button will clear the screen so that you can experiment some more.

In Chapter 11 you learned about using **Keep** and **Revert** from the **Paint** menu to protect your graphics. **DoMenu** makes it possible to write handlers to protect changes in

BEST COPY AVAILABLE

graphics on your card. Before changing any graphics, use **Keep** to "remember" the original graphics. After doing the animation, use **Revert**:

```
On mouseUp
   doMenu "Keep"
   doGraphics --handler to make graphics changes on the card
   doMenu "Revert"
end mouseUp
```

Unfortunately, this technique works only for graphics. Other changes you make on your card, such as moving buttons or hiding fields, are not affected by **Keep** and **Revert**.

Yet another way to use **doMenu** to protect your graphics is to make a copy of the card. For example, you might put this script on your graphics card:

```
on mouseUp
   doMenu "Copy Card"
   doMenu "Paste Card"
   set the name of this card to "temporary"
   doGraphics --handler to make graphics changes on the card
   doMenu "Cut Card"
end mouseUp
```

This technique creates the graphics on the copy of the card and then deletes the card when the graphics manipulations are completed.

Debugging and Testing

When you are working with **doMenu** you must be very careful to type the menu items exactly as they appear in the menus themselves. This means that if the menu item is followed by three periods, you must include those three periods in the **doMenu** command.

DoMenu won't work correctly if the appropriate menu item is not accessible. For example, if you type

```
choose browse tool
doMenu "Edit Pattern..."
```

nothing will happen because the **Paint** menu is not available when the Browse tool is selected. Similarly, if a menu item is "grayed out" or "dimmed," you will not be able to access it with **doMenu**.

When you use **doMenu** to create a new button, you can refer to it by the name "New Button." A new field doesn't have a default name. However, you can use the function **number** to set the name of the field:

```
on mouseUp
   doMenu "New Field"
   set the name of card field (the number of fields) to ¬
      "New Field"
end mouseUp
```

New HyperTalk Words in This Chapter

Commands	Functions
doMenu	random

Project Ideas

Experiment with **doMenu** from the Message box. Be sure you understand how to access a variety of menu items. This kind of practice saves you time when you begin to include **doMenu** commands in handlers.

Try writing some scripts to create new buttons and fields. Make use of your knowledge of **doMenu** and button and field properties.

Practice using random. You can begin by typing such commands as

```
random(500)
```

in the Message box. A number between 1 and 500 will appear. Be sure you understand what is happening when you use random.

Use **random** to place objects in random locations on the screen. For example, you could have a button that creates a new button or field and places it in a random location.

Explore the use of **doMenu** with graphics. Make use of some of the menu choices that you can't access in other ways.

Chapter 17. Message Passing

You have used **send** earlier in this book to test message handlers that you wrote. In this chapter you will learn more about using **send** to direct messages to specific objects.

In Chapter 11 you learned how to write your own message handlers. You saw that you could use these handlers to modularize your work. In the previous chapter, you made a Draw button that created a design made up of rectangles of random size filled with random patterns. You saw that you could test the individual handlers on the button by typing such commands as

```
send "drawRandomRect" to card button "Draw"
```

or

```
send "invertRect" to card button "Draw"
```

in the Message box. The **send** keyword allowed you to direct the message to the appropriate object on the card.

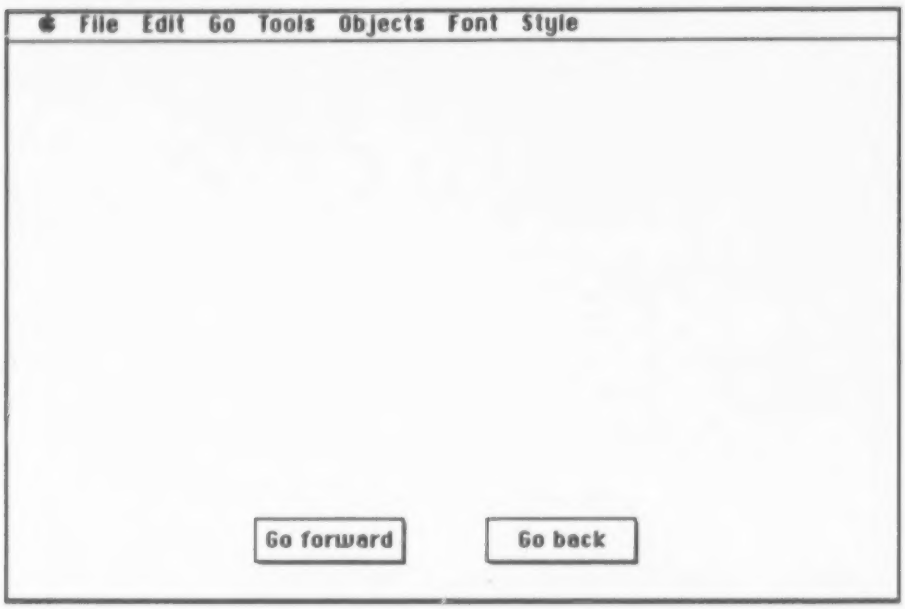
Send can also be used in handlers that you write to send messages to other *HyperCard* objects. **Send** takes the following form:

```
send "messageName [parameterList]" [to object]
```

Send sends a message directly to a particular object. It allows you to bypass any other handlers in the normal hierarchy.

Technical Note: You have seen that ***return, send, on, and end*** *are* keywords. *These words are predefined in HyperTalk, cannot be redefined, are not sent as messages, and, except for* ***send****, cannot be used in the Message box.*

To begin examining the use of **send** from handlers, create a card with two buttons, "Go back" and "Go forward":



At any point in time, you want only one of the buttons to be visible. That is, if you click on Go back, it disappears and Go forward appears. Similarly, if you click on Go forward, it disappears and Go back appears. You can put a script such as this on the Go forward button:

```
on mouseUp
   set the visible of me to false
   set the visible of card button "Go back" to true
end mouseUp
```

To illustrate the idea of message passing, however, you can write handlers that will pass messages between buttons. Begin by putting these two handlers on each button:

```
on hideButton
   set the visible of me to false
end hideButton

on showButton
   set the visible of me to true
end showButton
```

Technical Note: If you are using HyperCard 2.0, this might be a good time to display the Message Watcher window. (Type "show Message Watcher" in the Message box and check both of the boxes at the top of the window.)

Now you can type such instructions as

```
send "hideButton" to card button "Go forward"
```

and

```
send "showButton" to card button "Go back"
```

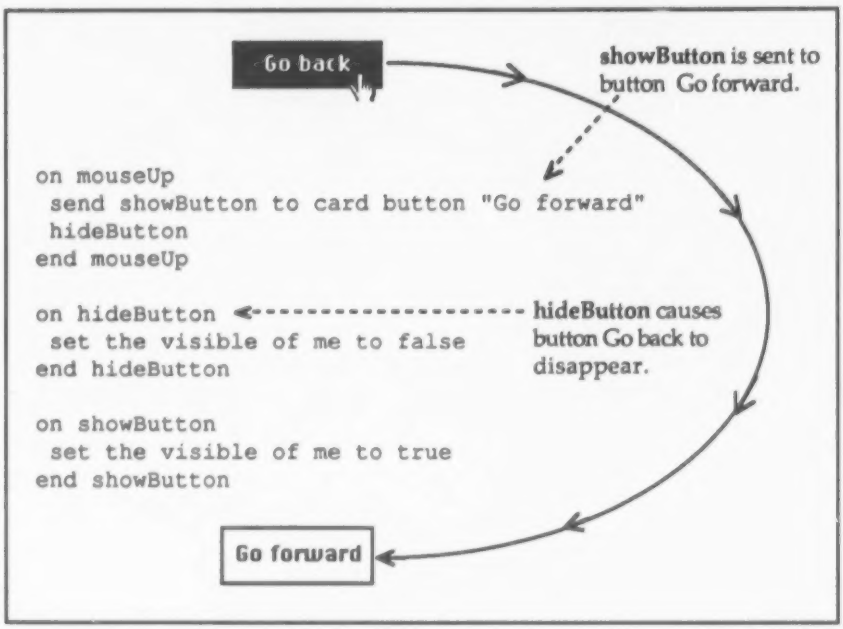
The messages are sent from the Message box to the buttons, and the buttons either become visible or invisible. You can, of course, have the buttons themselves send these messages. Here is a script for the Go back button that will accomplish this task:

```
on mouseUp
   send showButton to card button "Go forward"
   hideButton
end mouseUp
```

A similar script would be on the Go forward button:

```
on mouseUp
   send showButton to card button "Go back"
   hideButton
end mouseUp
```

When the mouse is clicked on the Go back button, a **showButton** message is sent to the Go forward button. This causes the Go forward button to become visible. The next command in the **on mouseUp** handler looks for a **hideButton** message, which it finds in the Go back handler. This causes the Go back button to be made invisible:

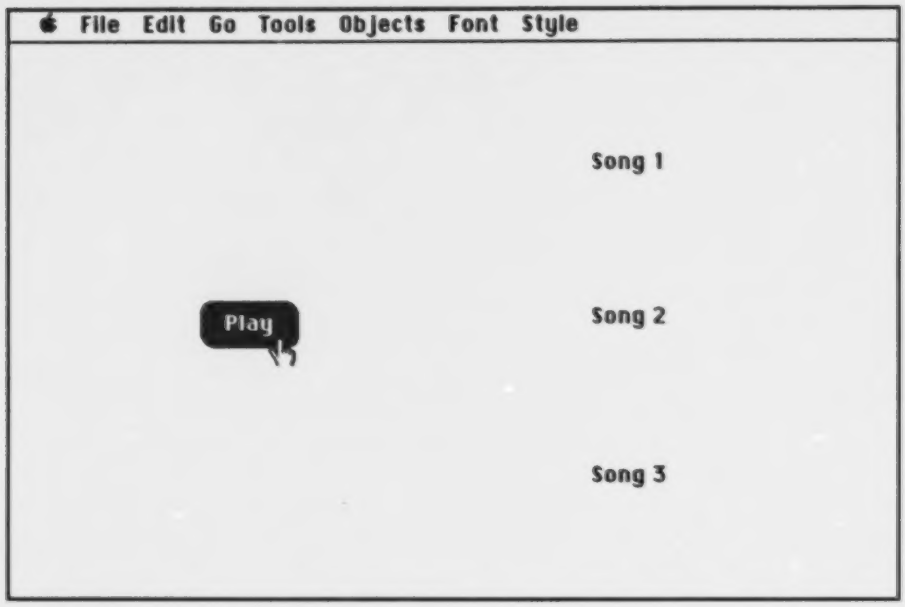


Note that both the **hideButton** and the **showButton** handlers use **me** to refer to the current object. Spend some time experimenting with this example. Do you understand what is happening?

Earlier in the book, you wrote scripts to play music. Suppose you have several pieces of music and would like to create a button that selects a random song to play. Begin by creating a "Play" button and three song buttons: "Song 1," "Song 2," and "Song 3." The script on the Play button should "activate" a random Song button:

```
on mouseUp
   set the hilite of card button "Play" to true
   send mouseUp to card button "Song 1"
   set the hilite of card button "Play" to false
end mouseUp
```

The first line of the script causes the Play button to be highlighted while the last line resets the **hilite** property to **false**. The **send** line of the script sends a **mouseUp** message to the Song 1 button. This script causes the Play button to remain highlighted while the song on the Song 1 button plays. Without this handler, the Play button would not remain highlighted during the song:



You have learned that typing

```
random(3)
```

produces a 1, 2, or 3. If you combine this random number with the word "Song" you can randomly select a button. This is accomplished with the concatenation symbol (&). Recall that you have to be careful to put a space after "Song":

```
"Song " & random(3)
```

You can also use the "concatenation with space" symbol (&&), and then you don't have to worry about putting a space after "song." To test this idea, type

```
"Song" && random(3)
```

in the Message box. You see Song 1, Song 2, or Song 3. Now add this statement to the script on the Play button so that it reads

```
on mouseUp
   set the highlight of card button "Play" to true
   send mouseUp to button ("Song" && random(3))
   set the highlight of card button "Play" to false
end mouseUp
```

If you add a song to each Song button, you will hear different songs each time you click on the Play button. (**Random** may pick the same number twice, so it is possible that you will hear the same song several times in a row.)

Send can be particularly useful when you want to pass messages between cards. Suppose you have a card "My Notes" with a field that appears, or pops up. When you click on a button called "Notes," a field appears. You want to be sure that the next

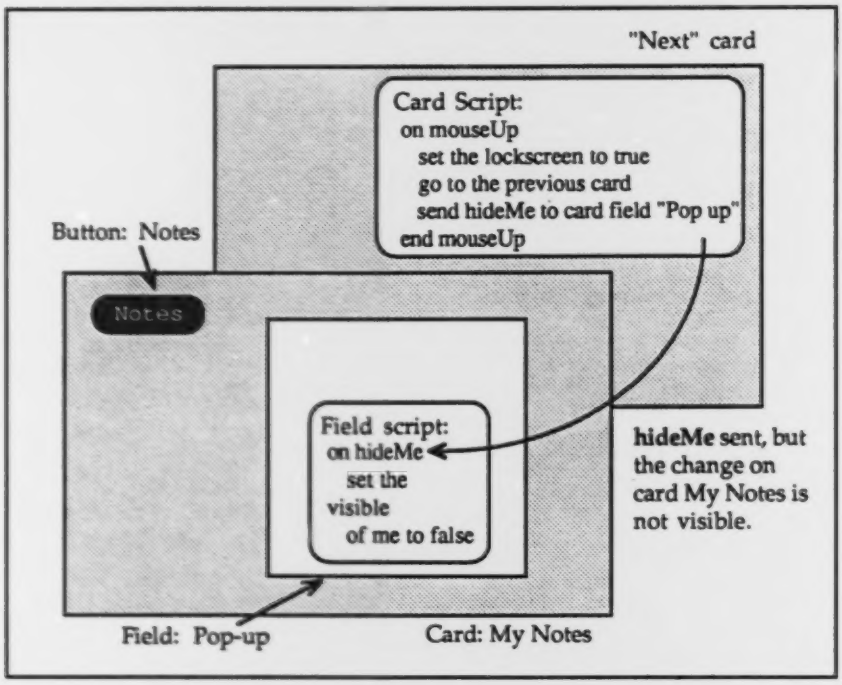
person who uses the stack does not see the pop-up field when this card is reached. Thus, you could put this handler on the field "Pop-up":

```
on hideMe
   set the visible of me to false
end hideMe
```

Then, the handler

```
on mouseUp
   set the lockScreen to true
   go to previous card
   send hideMe to card field "Pop up"
end mouseUp
```

can be added to the next card. (You will learn how to put handlers on cards in Chapter 18.) The **lockScreen** global property keeps the screen from changing. Thus, you do not see the My Notes card even though the **hideMe** message has been sent to it:



Debugging and Testing

When you begin passing messages among objects, there are many things that can go wrong. You can

- Make typographical errors
- Make syntax errors in your scripts
- Send messages incorrectly.

Because there are more possibilities for mistakes when you are sending messages, you need to use your best debugging skills. Check carefully for misspellings. Be sure you

have object names correct. Be sure you have included the correct message handlers on objects receiving messages.

Remember that you can test individual instructions from the Message box. Remember that you can use the comment mark (--) to run only a few lines of your handler at a time. If you are using *HyperCard* 2.0, make use of the debugging facilities, especially the Message Watcher window. See your documentation for details.

Most of all, take your time and think carefully when a script doesn't run successfully. Take notes so that you remember what you have tried. Take a break. Sometimes coming back to a problem after a period of time makes the error much easier to find.

New HyperTalk Words in This Chapter

Keywords	Global Property
send	lockScreen

Project Ideas

Spend some time practicing sending messages from one object to another. Can you predict what will happen? Do you understand what is happening when you use the **send** command?

Create a button that selects random graphics effects much as the example in this chapter plays random songs. For example, one button might fill the card with a pattern. Another might draw a series of circles. Yet another might spray-paint an area.

Extend the **play** example described in this chapter. When each song plays, have the button disappear and show an appropriate graphic. You might even include some animation with each song.

Chapter 18. Cards, Backgrounds and Stacks–Properties and Scripts

You have learned that *HyperCard* buttons and fields have a number of properties. You have also learned to attach scripts to buttons and fields. In this chapter you will learn about the properties of cards, backgrounds, and stacks, and you will learn how to write scripts for these *HyperCard* objects.

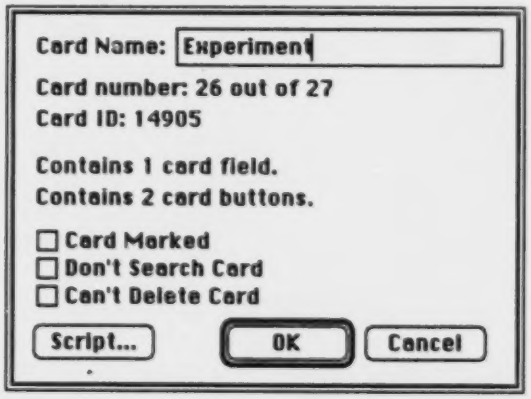
You have learned that there are a large number of properties for buttons and fields. These properties describe such things as the shape of the button or field, whether it is visible, and what type style is used in it. Cards, backgrounds, and stacks also have properties that define them.

Card and Background Properties

Cards and backgrounds have the following four properties:

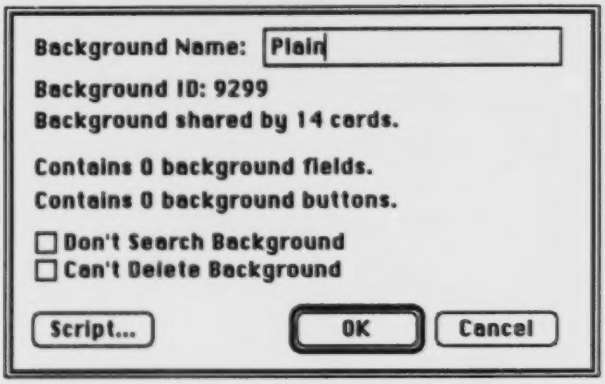
```
name
number
id
script
```

If you go to the **Objects** menu and select **Card Info**, you see



(If you are using a version of *HyperCard* before 2.0, the dialog box you see will be slightly different.)

If you select **Background Info**, you will see a similar dialog box:



You can get the **number** and ID of these backgrounds and cards by using instructions such as

```
the number of card "Experiment"
```

to see the **number** in the Message box, or

```
the background of card "Plain"
```

to see the ID of "Plain." As with buttons and fields, you can only examine these two properties—you cannot change them.

You can both examine and change the **name** of backgrounds or cards. The statement

```
the name of this background
```

lets you see the **name** of the current background in the Message box. You can also modify the **name** by using the **set** command:

```
set the name of this card to "Demo"
```

As with buttons and fields, you can examine or modify the scripts of cards and backgrounds by using the **script** property. For example, you can type

```
put the script of background "Introduction" into card field 1
```

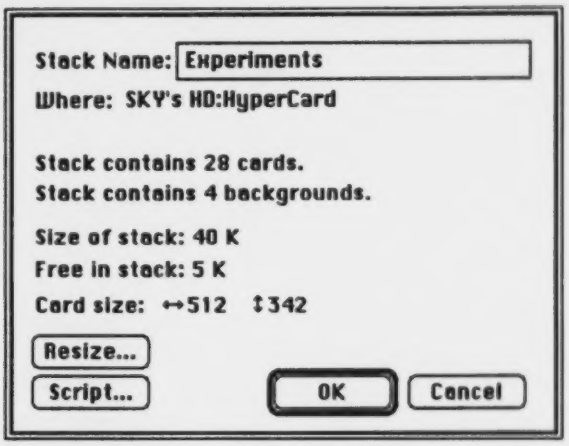
to have the background script appear in a card field, or you can type

```
set the script of the first card to empty
```

to remove the script from the first card.

Stack Properties

If you select **Stack Info**, you see a dialog box like this:



By examining this dialog box, you can see that stacks have somewhat different properties. These include

```
name
script
freesize
size
```

Like cards and backgrounds, stacks have a **name** and **script** property. These properties can be examined and modified in the same way that you examine and modify these properties for other *HyperCard* objects.

Stacks also have the properties **freesize** and **size**. These properties, like **number** and **ID**, can be examined but not changed. **Size** tells you how large your stack is and can have values from 4,096 bytes to 512 megabytes (534,592,893,840,329,290,847 bytes). The value of **size** changes as you add and delete things from the stack.

Freesize tells you how much free space there is in the stack. Perhaps you have noticed the menu choice **Compact Stack** under the **File** menu. As you work with stacks, they tend to get larger. You can make them as small as possible by using the **Compact Stack** option. **Freesize** tells you approximately how much space can be removed by compacting. After compacting a stack, the value of **freesize** is zero.

Scripting Cards, Backgrounds, and Stacks

So far you have written scripts only for buttons and fields. You can, of course, write scripts for other *HyperCard* objects. Scripts can be put into cards, backgrounds, and stacks. In each of the dialog boxes shown in this chapter, you can select **Script** to get to the script editor.

Perhaps you want the next card in the stack to appear when the user clicks anywhere on the card. Putting the handler

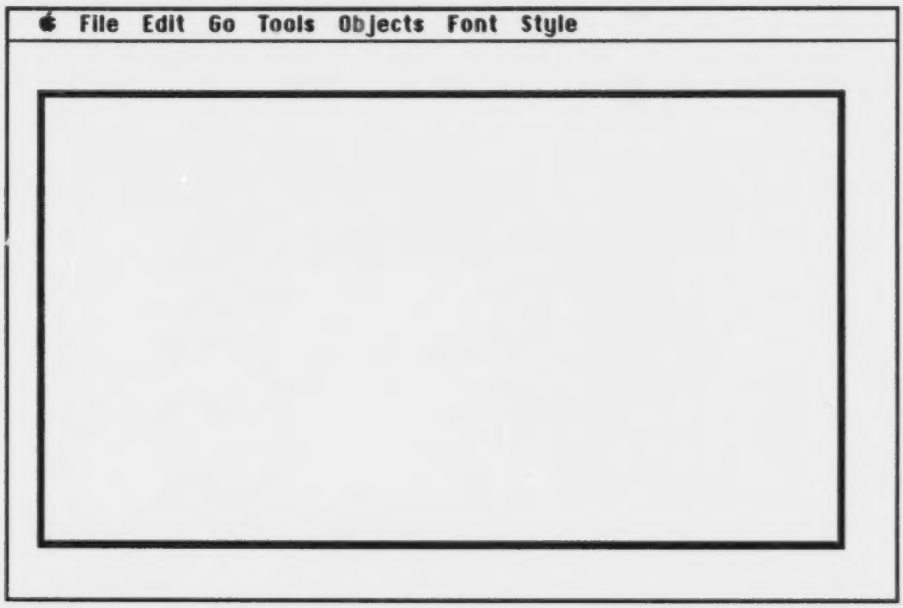
```
on mouseUp
   go to next card
end mouseUp
```

into the script editor of the card causes the next card to appear when the mouse is clicked anywhere on the card. Note that if there are buttons or fields on this card that have **mouseUp** handlers, the **mouseUp** message can be "used" by these buttons or fields. If the message is not used by a button or field, it falls through to the card. This technique of clicking anywhere on a card is useful when you want to display an introductory card at the beginning of your stack. No matter where users click, they will move to the beginning of your stack.

Suppose you want a "frame" to be slowly drawn on a card each time the user goes to that card. You can make use of the fact that an **openCard** message is automatically sent whenever you go to a new card. If you put this script on the card,

```
on openCard
   set dragSpeed to 20
   choose rectangle tool
   set centered to true
   set lineSize to 4
   drag from 250,180 to 480,310
   choose browse tool
end openCard
```

then when you go to the card, you see a rectangular frame appear:



You can then put an "erase" script on the card so that the frame disappears when you leave the card:

```
on closeCard
   choose select tool
   doMenu "Select all"
   doMenu "Clear picture"
   choose browse tool
end closeCard
```

The handler makes use of the fact that a **closeCard** message is sent whenever you leave a card.

BEST COPY AVAILABLE

Often a change in background is a visual indication to users that they are in a different part of a stack. You might want to add an auditory prompt as well. Thus, you might add a script such as the following to a background:

```
on openBackground
   play harpsichord c d e
end openBackground
```

This script makes use of the **openBackground** message that is sent each time you move to a *new* background.

Try putting the script on a background in your stack. Notice that the music plays each time you move to the background containing this script from a *different* background. If you move to another card with the same background, you will not hear the music. That is, whenever you change the background in use on the screen, an **openBackground** message is generated and music is played.

When you are experimenting with new ideas in *HyperCard*, you usually want to have access to the Message box and the Tool window. You may also want to have the Pattern window close at hand. You can make use of the fact that an **openStack** message is sent whenever you open a new stack. In the script editor for the stack, you can put

```
on openStack
   show message box
   show tool window
   show pattern window
end openStack
```

Then each time you open your experimental stack, the windows you want to use will be available. Because you can also specify the coordinates at which each window can be placed (see Chapter 8), you can even have your handler put the windows in the exact location you prefer:

```
on openStack
   show message box
   show tool window at 50,100
   show pattern window 400,100
end openStack
```

Placing Scripts

Because you can put a script on any *HyperCard* object, the question arises as to where a script should be placed. When should you put a script on the stack? When should it be on the background? There is no definite rule for where you place scripts. However, here are some "rules of thumb."

In general, you want scripts to be attached to objects that use those scripts. That is, if you want a button to cause a field to appear, the script for that button should be on the button. However, you might have a whole series of buttons that use the same script. For example, you might have a sound handler such as

```
on promptSound
   play "boing" "c4e g4e"
end promptSound
```

This handler can then be called from any button. For example, each button might contain

```
on mouseUp
   set the hilite of me to true
   promptSound
   set the hilite of me to false
end mouseUp
```

The **promptSound** handler then needs to be written only once. Because scripts further along in the message-passing hierarchy can be hard to find by someone reading your code, it is a good idea to include comments that indicate where the "remote" handlers are located:

```
on mouseUp
   set the hilite of me to true
   promptSound --promptSound is on the card
   set the hilite of me to false
end mouseUp
```

Suppose you have a group of cards sharing the same background that have identical scripts. For example, suppose you have a linear stack in which clicking anywhere on the card moves to the next card. Then it makes sense to put the script

```
on mouseUp
   go to next card
end mouseUp
```

on the background. Similarly, if a group of backgrounds all do the same thing, it may be sensible to put the script on the stack.

If you have scripts that serve as "tools" throughout your stack, the script should probably reside on the stack itself. For example, you might write a script that performs some complex computation or text manipulation. If this manipulation is done at a number of places in the stack, the stack is the appropriate location at which to place it.

Debugging and Testing

Accessing scripts on the card, background, and stack can be tedious if you have to pull down the **Objects** menu, select the appropriate menu choice, and then click on the Scripts button. There are keyboard shortcuts to get to the script editor for each of these objects. When you are in Browse mode, you can use

Command-Option-C	bring up the card script editor
Command-Option-B	bring up the background script editor
Command-Option-S	bring up the stack script editor.

Learning to use these shortcuts can save you a lot of time when working with scripts placed on a variety of objects. Recall that you can get to the script of a button by holding down Command and Option and clicking on the button. To get to a field script, you must hold down Command, Option, and Shift while clicking on the field.

When you are working with **openCard** and **closeCard**, it is easy to get into difficulty because every time you go to or leave the card, your script runs. (The same problem can occur when opening and closing backgrounds and stacks or when using the **lockMessage** property.) You can stop *any* script from running by using Command-period (Command-.).

Because it is easy for scripts to become "lost" when they are spread among many objects, it is important that you use comments to tell anyone (including yourself) who reads your scripts the location of other scripts you are using. This is particularly important when you use a handler on one object that is actually placed further along the message-passing hierarchy. That is, if you use a handler on a button that is actually defined on the card, there should be a comment in the button handler that tells the reader of your code where to find the card handler.

New HyperTalk Words in This Chapter

Card and Background Properties
name
number
ID
script

Script Properties
name
script
freesize
size

Keyboard Shortcuts	
Command-Option-C	bring up the card script editor
Command-Option-B	bring up the background script editor
Command-Option-S	bring up the stack script editor
Command-Option-click	bring up the button script editor of the button clicked on
Command-Option-Shift-click	bring up the field script editor of the field clicked on
Command-period	stop a script from running

Project Ideas

Develop a handler that makes use of **openStack** to use with the stack you use when experimenting with new *HyperCard* ideas. Arrange the screen in a way that is comfortable for you, showing the windows you generally use in the locations you prefer.

Experiment with visual or auditory prompts that occur when you move to a new card or new background. Be careful that you don't make these prompts so strong that they would be distracting to the user.

Put this handler on your current card:

```
on mouseUp
   beep 3
end mouseUp
```

In addition, put this handler on your current background:

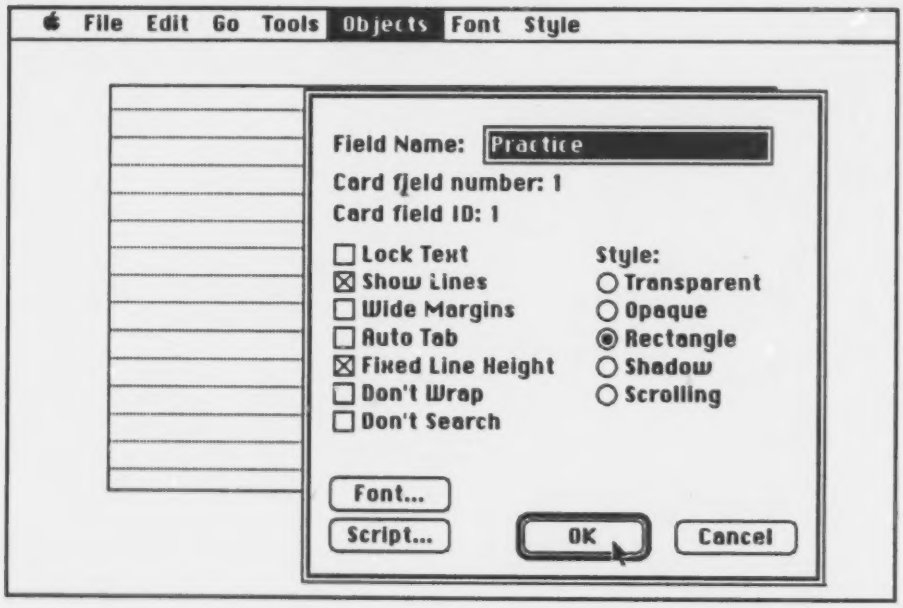
```
on mouseUp
   play "harpsichord" "c4q d e"
end mouseUp
```

What happens when you click over the card? Do you see that the **mouseUp** message is "used" by the card and thus never gets to the background? Try similar experiments with placing scripts on buttons, on fields, and on the card, background, and/or stack. Make sure you understand how the position of your mouse pointer can affect the results.

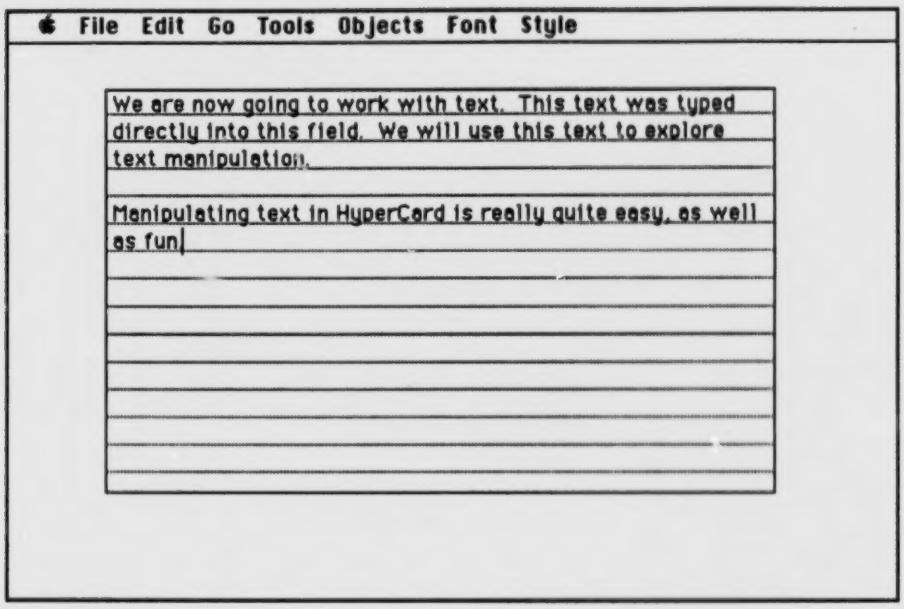
Chapter 19. Working With Text

You have worked with text in fields. You have seen chunks and chunk types (Chapter 3). In this chapter you will learn how to work with text in fields by using chunks, chunk types, chunk expressions, and several new commands and functions.

To begin working with text, create a card with a large field on it:



Enter some text in the field:


File Edit Go Tools Objects Font Style
We are now going to work with text. This text was typed directly into this field. We will use this text to explore text manipulation.
Manipulating text in HyperCard is really quite easy, as well as fun.

Chunks and Chunk Expressions

To work with text, you need to understand the concept of *chunks* and *chunk expressions*. A chunk is a

```
character (or char)
word
item
line
```

where these terms are defined as follows:

- A *character* is any single character, including numbers, punctuation, spaces, and return characters.
- A *word* is any collection of characters, including punctuation, that is separated by spaces.
- An *item* is any collection of characters, including punctuation, that is separated by commas.
- A *line* is any collection of characters that ends with a return character.

Technical Note: Spaces are characters in any source except words. Commas are counted as characters in any source except items. Return characters are counted as characters in items when counting all of the characters a field.

A chunk expression specifies a particular piece of text. For example, in the sample of field text given in the previous screenshot, if you type

```
character 6 of card field "Practice"
```

in the Message box, the letter "e" from "are" appears. If you type

```
word 5 of card field "Practice"
```

you see "to." If you type

```
item 2 of card field "Practice"
```

you see "as well as fun." Commas separate items, so the second item begins after the first comma in the fields. If you type

```
line 3 of card field "Practice"
```

you see "Manipulating text in *HyperCard* is really quite easy, as well as fun." Lines end with return characters. There is one return character at the end of the first paragraph. The blank line between the paragraphs contains a return character and thus counts as the second line of the field. The third return character is after the second paragraph. Thus, the second paragraph is selected. (Note that if you don't have the same number of return characters in your field, your practice example may not work correctly.)

In chunk expressions you can specify a *range* by using the preposition "to." For example, you can type

```
character 8 to 15 of card field "Practice"
```

and you see "now goin." Chunk expressions can become even more complex. For example,

```
character 3 to 5 of word 4 of line 3 of card field "Practice"
```

returns "per" from the word "*HyperCard.*" You can also use the words zero through ten, which are constants, in chunk expressions. The previous example would then become

```
character three to five of word four of line three of card
   field "Practice"
```

Notice that chunk expressions begin with the smallest part on the left and move to the largest part on the right. Also note that each part of the chunk expression is separated from the next by the preposition **of**. Recall that earlier you learned about *chunk types.* Chunk types are the plural of chunks—characters, words, items, and lines—and are used in statements such as

```
the number of characters in line 3 of card field "Practice"
```

Some Functions and Commands for Working With Text

There are also some functions that can assist you as you are working with text. You have seen the **length** function earlier in this book. **Length** works with chunks as well. If you type

```
the length of card field "Practice"
```

you see a number close to "206." (The number may vary if you have a different number of spaces and/or return characters.) Of course, you can get the length of any chunk, whether it is a word, item, or line, by typing expressions like

```
the length of line 1 of card field "Practice"
```

or

```
the length of word 6 of line 3 of card field "Practice"
```

The **offset** function allows you to locate the position of particular text within a larger body of text. Its syntax is:

```
offset (string1, string2)
```

Thus, if you type

```
offset ("going", card field "Practice")
```

you see "12" in the Message box. This tells you that the word "going" begins 12 characters from the beginning of the field.

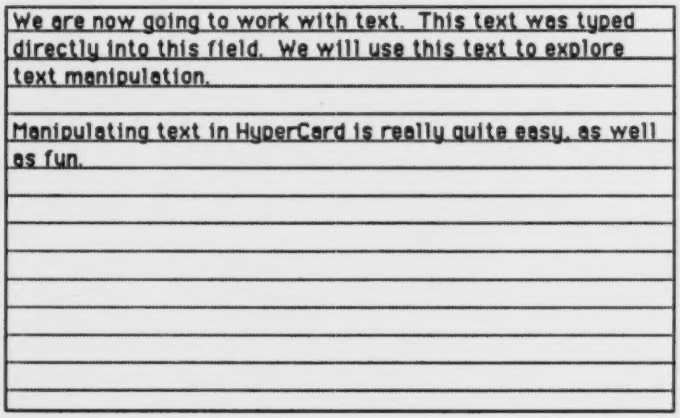
Perhaps you have used **Find** from the **Go** menu. There is also a **find** command that allows you to locate text in a field. Its syntax is:

```
find expression [in field fieldDescriptor]
find chars expression [in field fieldDescriptor]
find word expression [in field fieldDescriptor]
```

The first form of **find** matches the beginnings of words. The second form matches anywhere within words. The third form matches only complete words. If you use **find** to search card fields, it begins at the first card on your stack and goes to the last card of the stack. If a match is found, it shows the card on which the match occurs. If there is no match, then *HyperCard* beeps.

Technical Note: More recent versions of HyperCard *have added more features to the* **find** *command. You should see the documentation with your particular version of* HyperCard *to determine which features you have available. It is important to read this documentation carefully because there are so many variations on the way* **find** *can work.*

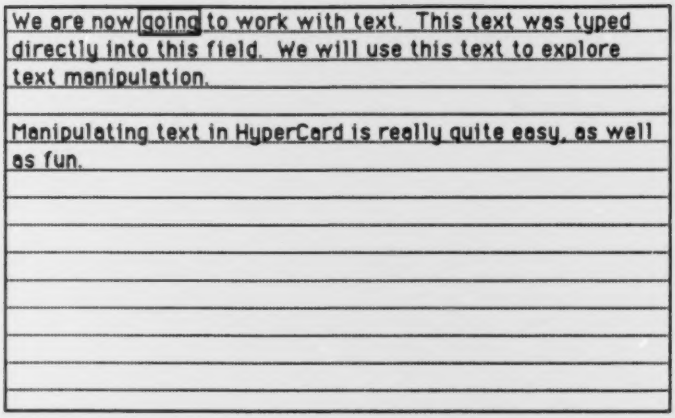
Here are some examples. Assume that you have only the card field shown here in your stack:


We are now going to work with text. This text was typed directly into this field. We will use this text to explore text manipulation.
Manipulating text in HyperCard is really quite easy, as well as fun.

If you type

```
find "going"
```

a rectangle appears around the word "going":


We are now going to work with text. This text was typed directly into this field. We will use this text to explore text manipulation.
Manipulating text in HyperCard is really quite easy, as well as fun.

Similarly, if you type

```
find chars "go"
```

the "go" will be outlined, but if you type

```
find word "go"
```

HyperCard will beep because a separate word "go" does not appear in the text. If you type

```
find chars "as"
```

the "as" in "was" will be boxed. Remember, **find** alone matches the beginnings of words; **find chars** matches characters anywhere in the text, and **find word** matches only whole words.

There are four functions that allow you to use HyperTalk to get the value of the expression located by using **find**. They are:

```
the foundText
the foundChunk
the foundLine
the foundField
```

You cannot work with these functions in the Message box because the rectangle around the text you locate with **find** disappears when you begin to type in the Message box. To explore the use of these functions, put a handler on the card. For example, if you put

```
on mouseUp
   find "text"
   put the foundText
end mouseUp
```

and click anywhere on the card, you see "text" in the Message box. At this point in your work with HyperTalk, the most valuable of these four functions is **foundField**. If you type

```
on mouseUp
   find "text"
   put the foundField
end mouseUp
```

you see

```
card field "Practice"
```

in the Message box. Note that **foundChunk** returns an expression such as

```
char 32 to 35 of card field 1
```

while **foundLine** returns an expression such as

```
line 1 of card field 1
```

Technical Note: If you are using a version of HyperCard *before 1.2, the find functions described here are not available.*

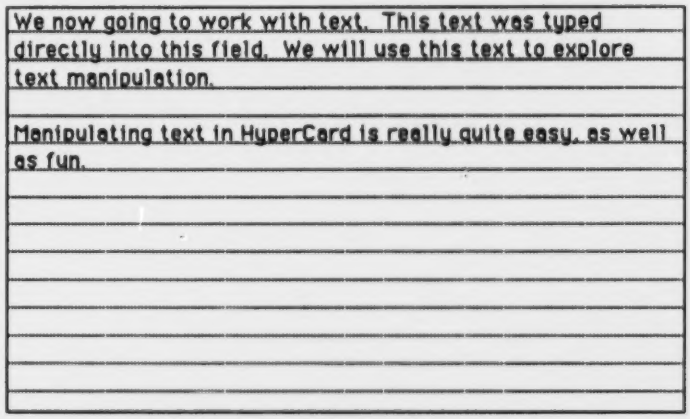
The **delete** command allows you to delete a specified chunk. The syntax of **delete** is:

```
delete chunk [of container]
```

Thus, if you type

```
delete word 2 of card field "Practice"
```

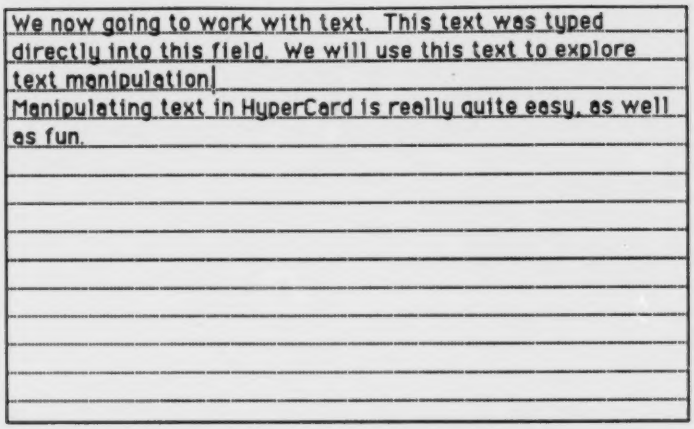
the word "are" disappears:


We now going to work with text. This text was typed directly into this field. We will use this text to explore text manipulation.
Manipulating text in HyperCard is really quite easy, as well as fun.

If you type

```
delete line 2 of card field "Practice"
```

you see


We now going to work with text. This text was typed
directly into this field. We will use this text to explore
text manipulation
Manipulating text in HyperCard is really quite easy, as well
as fun.

Note that line 2 in the field Practice was the blank space between paragraphs. Because this space was actually an invisible carriage return, it was considered to be a line and was deleted.

You can use the **put** command to put text into a field. Recall that the syntax for **put** is:

```
put expression [preposition container]
```

"Preposition" can be "into," "before," or "after." Thus, typing

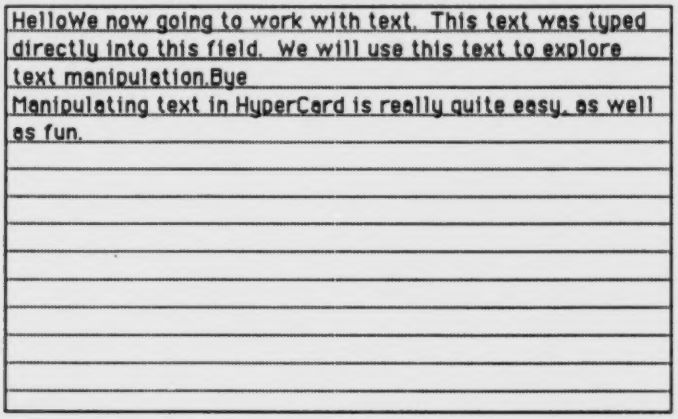
```
put "Hello" before line 1 of card field "Practice"
```

inserts the word "Hello" at the beginning of the text. If you want a space after "Hello," you can use "Hello "—the word "Hello" with a space after it.

If you type

```
put "Bye" after line 1 of card field "Practice"
```

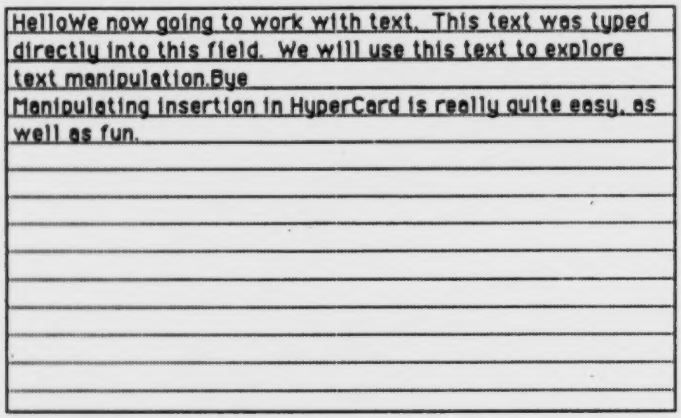
the word "Bye" appears after the word "manipulation":


HelloWe now going to work with text. This text was typed
directly into this field. We will use this text to explore
text manipulation.Bye
Manipulating text in HyperCard is really quite easy, as well
as fun.

You can use **put** to replace parts of the text. If you type

```
put "insertion" into word 2 of line 2 of card field "Practice"
```

the word "text" is replaced with the word "insertion":

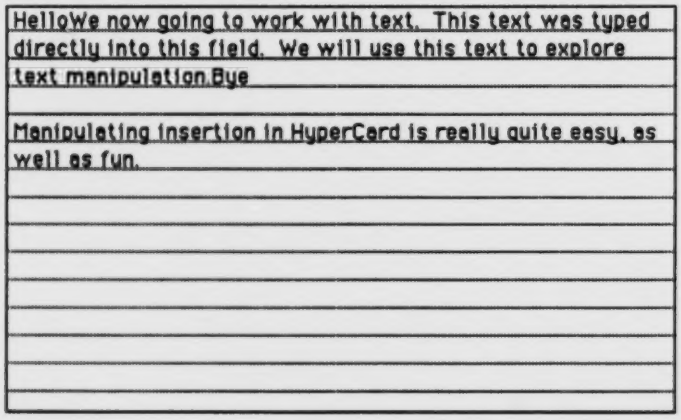

HelloWe now going to work with text. This text was typed directly into this field. We will use this text to explore text manipulation.Bye
Manipulating insertion in HyperCard is really quite easy, as well as fun.

Some Constants and Operators for Working With Text

There are some other constants that can be helpful in working with text. You have seen the constant **Return** earlier. If you type

```
put Return after line 1 of card field "Practice"
```

a return character is inserted between the first paragraph (*HyperCard* line) and the second paragraph (*HyperCard* line).


HelloWe now going to work with text. This text was typed directly into this field. We will use this text to explore text manipulation.Bye
Manipulating insertion in HyperCard is really quite easy, as well as fun.

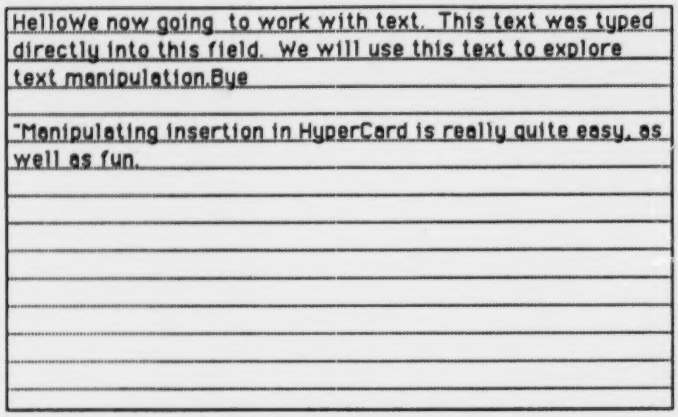
If you type

```
put Quote before word 1 of line 3 of card field "Practice"
```

a double quotation mark appears before the first word of the third line. If you type

```
put Space after word 3 of line 1 of card field "Practice"
```

an extra space will appear after the word "going." Note that HyperTalk counts "HelloWe" as one word since there is no space between the two words:


HelloWe now going to work with text. This text was typed directly into this field. We will use this text to explore text manipulation.Bye
"Manipulating insertion in HyperCard is really quite easy, as well as fun.

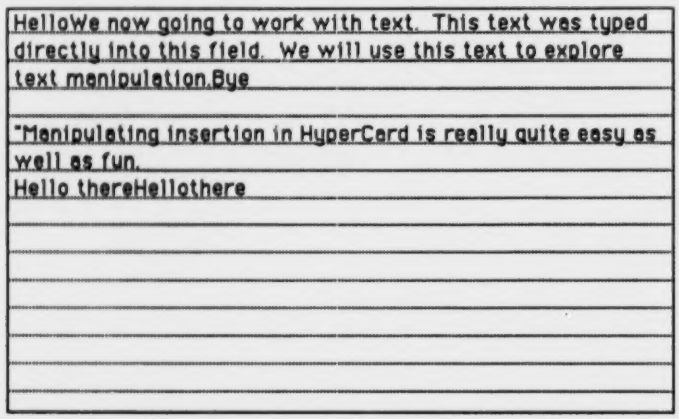
You can combine text by using the concatenation operators. You have seen concatenation with the spaces operator (&&) in Chapter 17. If you type

```
put "Hello" && "there" after card field "Practice"
```

you see the words "Hello there" separated by a space. If, however, you type

```
put "Hello" & "there" after card field "Practice"
```

you see "Hellothere." There is no space between the words:


HelloWe now going to work with text. This text was typed directly into this field. We will use this text to explore text manipulation.Bye
"Manipulating insertion in HyperCard is really quite easy as well as fun.
Hello thereHellothere

There are three operators that allow you to check for particular chunks of text. Each of these operators returns **true** or **false**. If you type

```
"hypertalk" contains "hyper"
```

you see **true** in the Message box. Similarly, if you type

```
"hyper" is in "hypertalk"
```

you see **true**. If you type

```
"hyper" is not in "hypertalk"
```

you see **false**. These operators can be used with the text in a field. For example, if you type

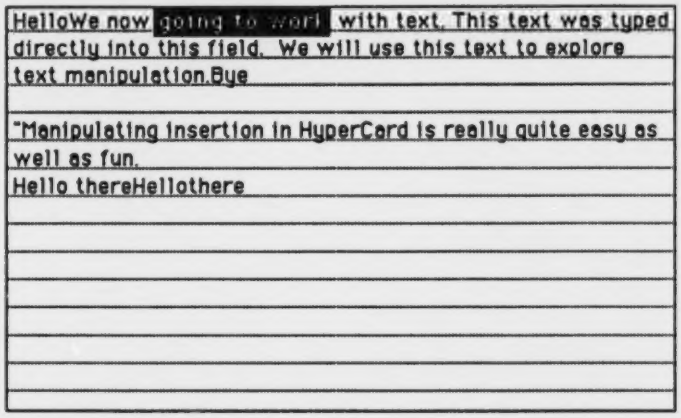
```
"insertion" is in card field "Practice"
```

true appears in the Message box. Similarly, if you type

```
"Bye" is in line 1 of card field "Practice"
```

you see **false** in the Message box. Later, in Chapter 21, you will learn how to use these operators to check for particular text in fields.

You have learned about two containers in your work with HyperTalk thus far. You learned that the Message box is a one-line container. You can type HyperTalk statements in the Message box and then run them. You learned that fields are both objects and containers. You can put text into the field. In this case, the field serves as a container for the text. There is a third container called **the selection. The selection** is the currently highlighted text. Thus, in the following example, the chunk "going to work" is currently in **the selection:**


HelloWe now going to work with text. This text was typed
directly into this field. We will use this text to explore
text manipulation.Bye
"Manipulating insertion in HyperCard is really quite easy as
well as fun.
Hello thereHellothere

In order to work with **the selection,** you need to learn about the **select** command. There are four forms of the **select** command:

```
select objectDescriptor
select [preposition] chunkExpression of fieldDescriptor
select [preposition] text of fieldDescriptor
select empty
```

You can use the Message box to examine each of these forms of **select.** The first form of **select** can be used with any object. If you type

```
select card field 1 of this card
```

the Field tool is selected and you see the dotted lines around the first field of the card, just as if you had clicked on it.

The other forms of **select** are used with text in a field. If you type

```
select before line 2 of card field "Practice"
```

the cursor appears at the end of the first paragraph, after the word "Bye." If you type

```
select after text of card field "Practice"
```

the insertion point is placed at the end of the text in the field. If you omit the preposition and type

```
select line 1 of card field "Practice"
```

the entire first paragraph (line/chunk) is selected. Finally

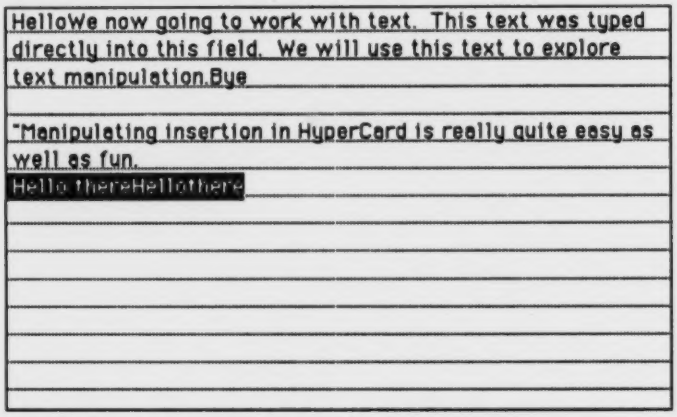
```
select empty
```

removes the highlighting in the field.

The selection can then be used to determine what text has been selected. Thus, if you put this script on the card,

```
on mouseUp
   select line 4 of card field "Practice"
   put the selection
end mouseUp
```

and click on the card, you see


HelloWe now going to work with text. This text was typed directly into this field. We will use this text to explore text manipulation.Bye
"Manipulating insertion in HyperCard is really quite easy as well as fun.
Hello thereHellothere

and "Hello thereHellothere" appears in the Message box.

There are also four functions that return information about the current selection. They are:

```
the selectedText
the selectedChunk
the selectedLine
the selectedField
```

These four functions return information about the value of **the selection**. For example, if you modify the card script as follows,

```
on mouseUp
   select line 4 of card field "practice"
   put the selectedText
end mouseUp
```

you see

```
Hello thereHellothere
```

in the Message box. If you replace **selectedText** with **selectedChunk**, you see something like

```
char 216 to 236 of card field 1
```

Using **selectedLine** gives

```
line 4 of card field 1
```

while using **selectedField** gives

```
card field 1
```

Technical Note: If you are using a version of HyperCard before 1.2, the select functions described here are not available.

A Note on HyperText

Earlier in this book (Chapter 10) you learned a way to create a hypertext-like environment using buttons to create "hot" words or phrases. If you are using *HyperCard* 2.0 or above, you can create more useful hypertext environments that will work even in scrolling fields.

First, select the word or words that you want to link to another location in your stack. Assign them the text style "group." You can type something like

```
set the textStyle of words 3 to 5 of card field 1 to "group"
```

Your word or words are now ready to create a link. Click in your grouped text. Then type

```
the clickText
```

The grouped text appears in the Message box. **ClickText** is a function that returns the word or group of words most recently clicked. Its syntax is:

```
the clickText
clickText ()
```

Suppose your grouped text were "HyperTalk is fun." Then, you might put a script on the field that reads

```
on mouseUp
if the clicktext = "HyperTalk is fun" then
   go to next card
end if
end mouseUp
```

Note that the field must be locked in order for this to work.

There is still one problem with this hypertext environment. The user has no way of knowing which text in your fields is linked to other parts of your stack. That problem is solved with the following commands:

```
show groups
hide groups
```

After you type **show groups**, grouped text has a wide gray line under it. This remains in effect until you leave *HyperCard* or type **hide groups.**

Debugging and Testing

It is easy to forget that spaces and return characters are "real" characters when you are working with text. It is especially easy to forget about return characters. Remember that every time you press the Return key, you are entering a return character into the text. Every time you see a blank line in a field, it contains a single return character (and possibly some spaces as well).

You need to think carefully about using numbers when working with text in fields. For example, typing

```
put "1" is in "21 22 23 24" into the Message box
```

causes **true** to appear in the Message box. It is easy to think of the string "21 22 23 24" as four numbers, but *HyperCard* sees it as a string of characters: 2 1 space 2 2 space 2 3 space 2 4.

New HyperTalk Words in This Chapter

Functions	
clicktext	foundField
offset	selectedText
foundText	selectedChunk
foundChunk	selectedLine
foundLine	selectedField

Commands
delete
find
hide groups
select
show groups

Chunks
character (or char)
item
list
word

Text Operators
&&
&
contains
is in
is not in

Constants
quote
return
space

Project Ideas

There are many new ideas presented in this chapter. Spend time experimenting with various chunk expressions. Be sure you understand how to insert and delete any character, word, item, or line in a field. Work with **find** and **offset**. Work with the functions associated with **find**. Experiment with the various text operators. Experiment with **select**, **the selection**, and the functions that work with **select**. You will be able to make use of all of these new ideas only if you are comfortable with using them.

Chapter 20. Interactive Programming

All of the action taken by your HyperTalk scripts so far has been completely planned by you. In this chapter, you will learn how to accept input from the user and make use of that input to modify the actions taken by your scripts.

No doubt you have used many computer programs that ask you questions and request a response from you. Sometimes you are asked to type a response, such as your name. Sometimes you merely need to click on a button. There are two ways you can use HyperTalk to interact with the user.

Ask and Answer

The first interactive command is **answer**. The syntax of **answer** is:

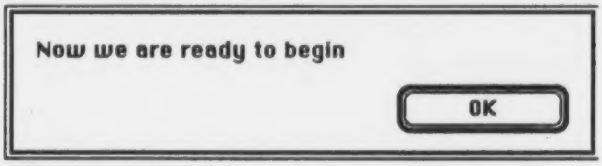
```
answer string1 [with string2 [or string3] or string4]]]
```

"String1" is the question to be asked; "string2," "string3," and "string4" are the possible replies.

If you use the simplest form of **answer**, a dialog box with just an OK button appears. That is, if you type

```
answer "Now we are ready to begin"
```

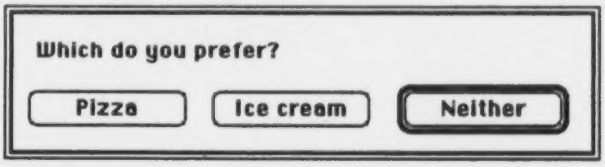
you see



If you look carefully at the syntax of **answer**, you see that you can have up to three responses, for example,

```
answer "Which do you like?" with "pizza" or "ice cream" or ¬
   "neither"
```

You see



Note that the button on the far right is always highlighted. Also note that "question" and "reply" are limited in length by the size of the dialog box and the size of the

buttons. That is to say, there is no "word wrap" allowed for the question and you cannot have multiline buttons.

Recall that you have seen three kinds of containers: the Message box, fields, and **the selection**. Each of them holds information. **It** is a special container. The response you give to an **answer** command automatically goes into a container called **it**. You will learn more about **it** in Chapter 23.

If you click on Ice cream in the previous dialog box, and then type

```
it
```

in the Message box, you see

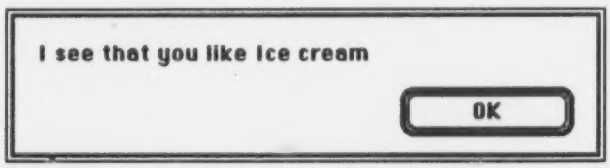
```
Ice cream
```

The result of **answer** is always automatically placed in the container **it**.

You can use the value of **it** to create another dialog box. In this case you might type

```
answer "I see that you like"&& it
```

Recall from Chapter 17 that && is used to concatenate two strings. You see



Notice that the "Ice" in Ice cream is capitalized in the response and that there is no period after it. You have to think carefully about what you want to include in your response box. In Chapter 21, you will learn how to test the result of an **answer** command and act on the result.

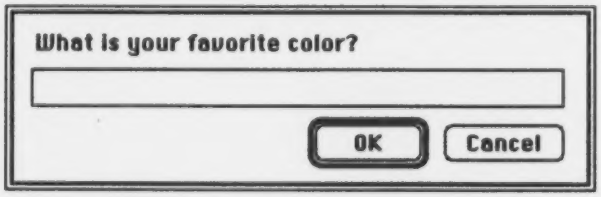
The other interactive command in HyperTalk is **ask**. It is quite similar to **answer**. The syntax of **ask** is:

```
ask string1 [with string2]
```

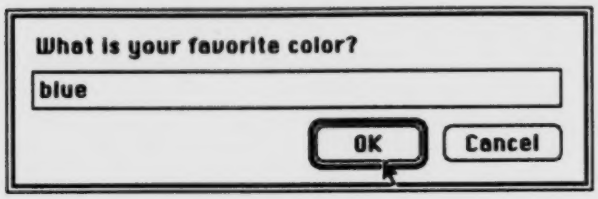
Here, "string1" is the question to be displayed, while "string2" is the default string. Like **answer**, the response from **ask** is put into **it**. For example, if you type

```
ask "What is your favorite color?"
```

you see



You can then type your answer.



The response "blue" is now in it. You might then type

```
answer "I like" && it && "too!"
```

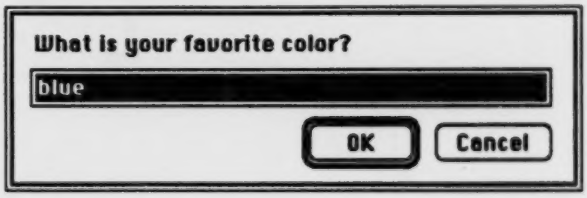
You then see

Note that if you click the choice "OK" in the dialog box containing "What is your favorite color?" without typing anything *or* if you click the choice "Cancel," it is empty.

When you use a second string with **ask**, that default string automatically appears in the text box. The user may then click on OK to continue or may type a different response into the box. For example

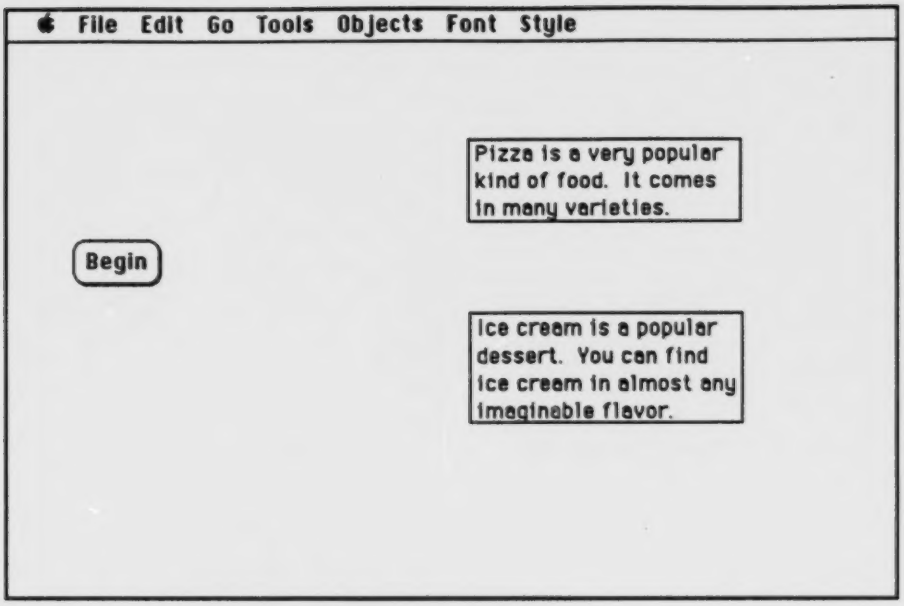
```
ask "What is your favorite color?" with "blue"
```

causes the dialog box to appear with "blue" already in it:



Using Ask and Answer

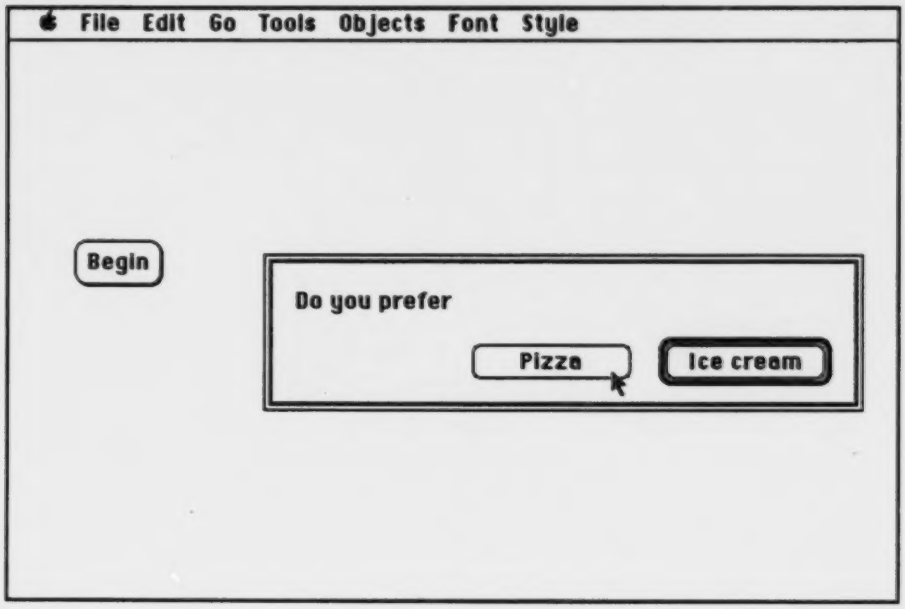
You can use **ask** and **answer** to get information from the user and can then use it to change the path through your stack. Suppose you have a card that looks like this. The top field is named "Pizza" and the bottom field is named "Ice cream":


File Edit Go Tools Objects Font Style
Pizza is a very popular kind of food. It comes in many varieties.
Begin
Ice cream is a popular dessert. You can find ice cream in almost any imaginable flavor.

The script on the Begin button might read as follows:

```
on mouseUp
   answer "Do you prefer" with "Pizza" or "Ice cream"
   show card field it
end mouseUp
```

Hide both of the card fields and click on the Begin button. You see



If you click on Pizza, the Pizza field appears. If you click on Ice cream, the Ice cream field appears. Recall that the result from **answer** is automatically put into **it**. Thus, when HyperTalk encounters

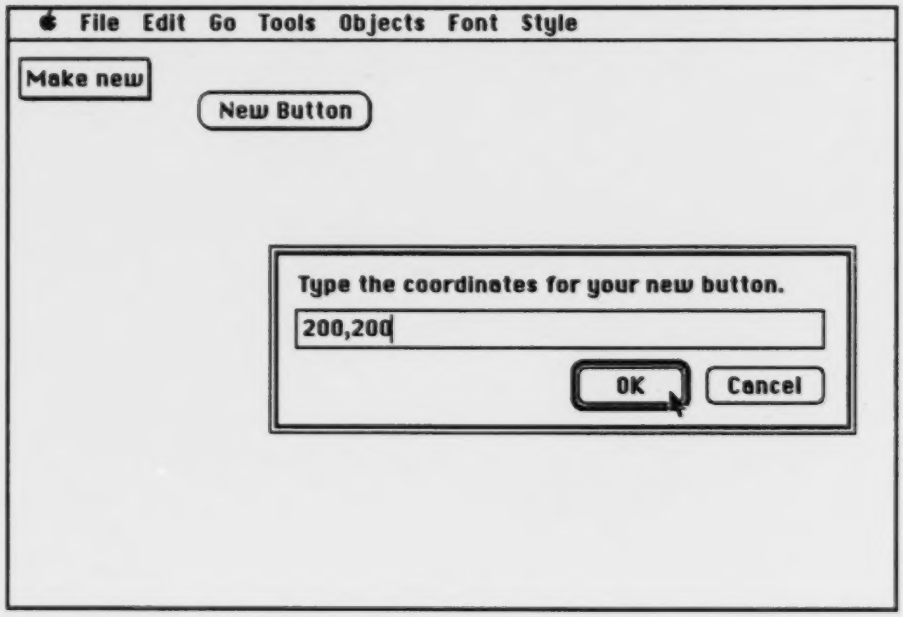
```
show card field it
```

it uses whichever value was put into **it** to determine which field to display.

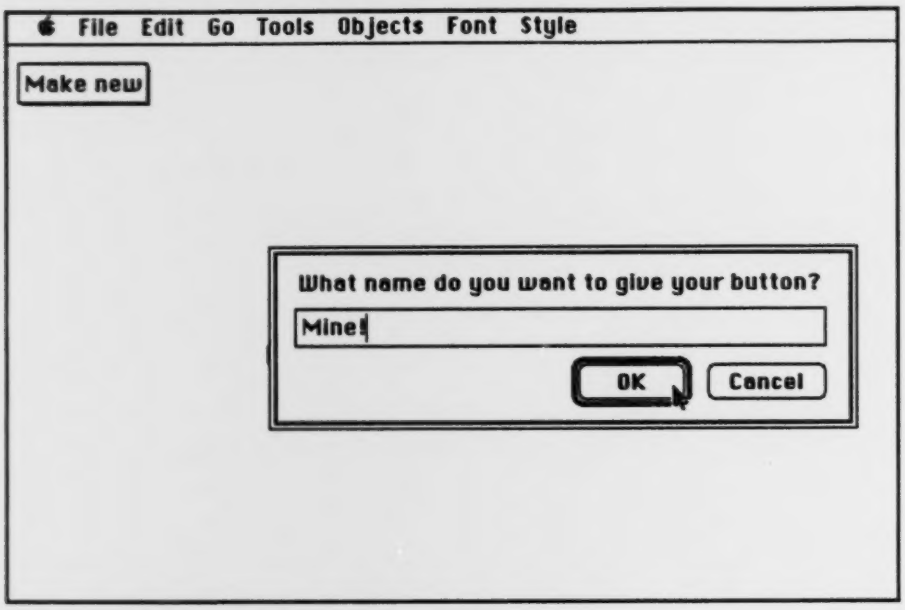
Earlier you learned to use **doMenu** to access menu commands. You can combine what you have learned about **doMenu** and what you have learned about interactive commands to allow the user to create a customized button. Create a button called "Make new." Put the following script on the button:

```
on mouseUp
   doMenu "New button"
   set the location of card button "New Button" to 160,60
   ask "Type the coordinates for your New Button."
   set the location of card button "New Button" to it
   ask "What name do you want to give your button?"
   set the name of card button "New Button" to it
   choose browse tool
end mouseUp
```

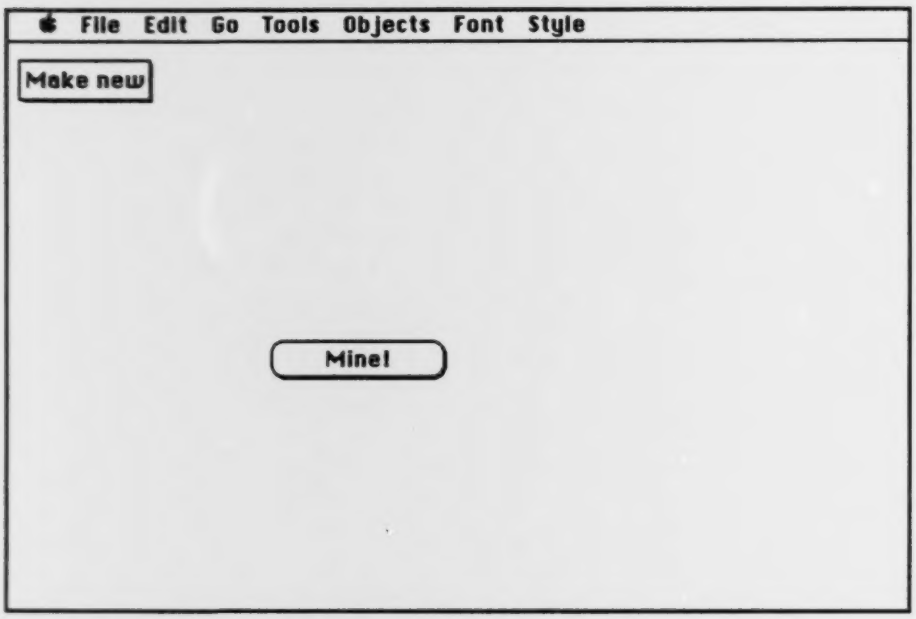
First, a new button appears near the top of the card:



If you then enter the coordinates 200, 200, you see



If you enter "Mine!" for the name of your button, you see



Note that each time you click on the Make new button, it will create another button. This button could be used as a tool to create a group of buttons that you want to use on a particular card.

Debugging and Testing

Recall that when you use **doMenu** to create a new button, you can always refer to the most recently created button by its name "New Button." However, when you create a new field, there is no default name. In that case, you must use "the number of fields" or "the last card (or background) field" to refer to that field:

```
on mouseUp
   doMenu "new field"
   set the name of card field (the number of card fields) to¬
      "New Field"
   choose browse tool
end mouseUp
```

Technical Note: The parentheses around "the number of card fields" is not necessary. Rather, they help make the script more readable.

The container **it** is a *local variable*. You will learn more about the containers called variables in Chapter 23. For now, you need to understand that **it** keeps its value only while the script containing **ask** or **answer** is running. That is, if you type

```
put it
```

in the Message box after you have used the Make new button to create a new button, no value will appear. However, if you insert the command "put it" just before the "end mouseUp" in the script for the Make new button, the name you type for the button will appear in the Message box. It has a value while the script is running, but it does not keep that value when the script is finished running.

New HyperTalk Words in This Chapter

Commands	Containers
ask	it
answer	

Project Ideas

Use **ask** and/or **answer** to create a script that carries on a conversation with the user. Ask the user for his or her name, favorite color, age, favorite music, and so forth. You might consider using a field to display your responses to the answers.

Use **doMenu** and **ask** or **answer** to create new objects. Write scripts to create particular kind of buttons or fields. Can you create a button utility card and a field utility card that lets you create custom buttons you can then copy and paste elsewhere?

Chapter 21. Conditionals

In this chapter you will learn how to instruct HyperTalk to make decisions. That is, you will learn to write instructions that carry out different actions depending on the results of a particular condition.

In your work with **ask** and **answer** you have seen that you can use the value of **it** to cause different actions to occur depending on which button the user clicks. When you use **ask** and **answer**, your options are limited by the syntax of those commands. In this chapter you will learn a more flexible way to have HyperTalk carry out actions that are dependent on such things as the user's response. Such actions are called *branching*. All computer programming languages include the ability to *branch*. In HyperTalk, the keyword used to do branching is **if**.

Single Statement If-Then

In its simplest form, the **if** statement is written

```
if condition then statement
```

or

```
if condition
then statement
```

Note that both **if** and **then** are keywords in HyperTalk.

To begin exploring the use of the simple form of the **if** statement, create a button and put the following handler on it:

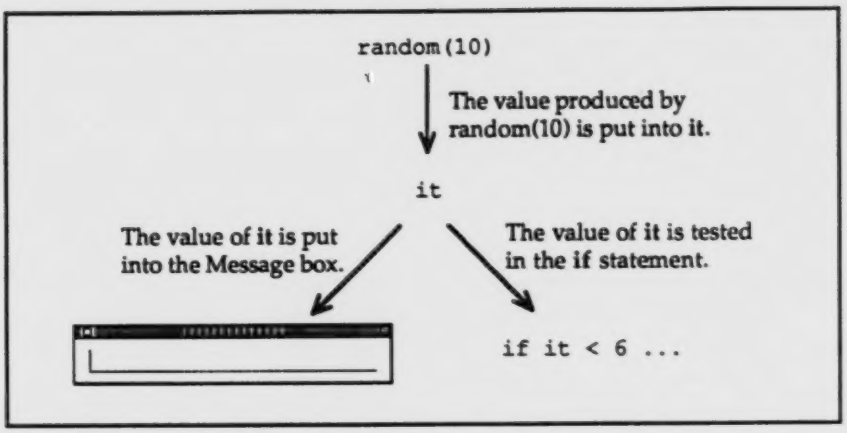
```
on mouseUp
   if random(10) < 6 then beep
end mouseUp
```

Recall that **random(10)** returns a value from 1 to 10. This number is compared with 6. If the number is less than 6, the *condition* [random(10) < 6] is **true**; if the number is 6 or greater, the *condition* is **false**. The HyperTalk code appearing after the **if** and before the **then** is called a *condition* and must produce either **true** or **false**.

If you now click on your new button, sometimes you will hear a beep and sometimes you won't. The result depends on the number picked by **random**. You can write this handler in such a way that you can also see the number picked by **random**:

```
on mouseUp
   put random(10) into it
   put it into Message box
   if it < 6
   then beep 3
end mouseUp
```

Note that in this case the value of **random(10)** is put into **it**. Recall that **it** is a local variable or container. (You have only seen **it** used with **ask** and **answer** and will learn more about **it** in Chapter 23.) **It** can then be used to get that value into the Message box and to test the random number picked:


random(10)
The value produced by random(10) is put into it.
it
The value of it is put into the Message box.
The value of it is tested in the if statement.
if it < 6 ...

Clicking on the button now lets you see and hear the result of **random**.

Multiple Statement If-Then

Sometimes you want more than one action to occur after you test a value. For example, you might like to have the beep that occurs randomly be combined with moving to the next card in your stack. To do this, you need the multiple statement form of **if**:

```
if condition then
   statement list
end if
```

Modify your script as follows:

```
on mouseUp
   if random(10) < 6 then
      beep
      go to next card
   end if
end mouseUp
```

Notice that this form of the **if** structure allows you to have any number of statements after the **if**. Also note that you must have **end if** to tell HyperTalk when you have reached the end of the statements to run based on the condition after the **if**.

Take a moment to think about how this handler works. If the number selected by **random** is less than 6, you hear a beep and go to the next card. However, if the number picked is 6 or greater, nothing happens. That is, when the condition after **if** produces **false**, all of the statements between the **if** and the **end if** are ignored.

To examine this idea a bit further, add a statement after the **end if**, for example,

```
on mouseUp
   if random(10) < 6 then
      beep
      go to next card
   end if
   put "After end if" into the Message box
end mouseUp
```

Now click on the button several times. Sometimes it moves to the next card. Sometimes it doesn't. However, "After end if" always appears in the Message box. Be sure you

understand what is happening here. Take some time to experiment with these new ideas.

Single Statement If-Then-Else

The previous button script is not particularly useful. Sometimes it moves the user to the next card. Sometimes it doesn't. More likely, you would want to move to one card if the number picked were less than 6 and to a different card if the number picked were 6 or greater. Or perhaps you'd like to announce the movement to the next card with a sound if the number were less than 6 or a visual effect if the number were 6 or greater. To accomplish this, you need to use the **if-then-else** form of the **if** statement. In its simplest form it is

```
if condition then statement else statement
```

or

```
if condition
then statement
else statement
```

Thus your handler might look like this:

```
on mouseUp
   if random(10) < 6
   then beep
   else visual effect checkerboard
   go to next card
end mouseUp
```

This handler moves to the next card in the stack no matter what number is picked. The "go to next card" statement is *after* the **if** statement now. In the previous section, you saw the multiple statement **if-then.** That structure ends with **end if.** However, here you are using the single statement **if-then-else,** so there is no **end if** to mark the end of the **if** structure.

Multiple Statement If-Then-Else

There is also a multiple statement form of the **if-then-else** structure. It looks much like the multiple statement form of the **if-then**:

```
if condition then
   statement list
else
   statement list
end if
```

Suppose you want to use the choice of a random number not only to produce different effects but also to go to different cards. Your script might look like this:

```
on mouseUp
   if random(10) < 6 then
      beep
      go to card "After beep"
   else
      visual effect checkerboard
      go to card "After visual"
   end if
end mouseUp
```

Now the user sees a different card depending on the random number picked.

You have seen four forms of the **if** structure. This can be very confusing for a beginner. In fact, you never need to use the single-statement forms of **if-then** and **if-then-else**. Even if you have only one statement in each part of the **if**, you can use the multiple statement form. For example, you can write

```
on mouseUp
   if random(10) < 6
   then
      beep
   else
      visual effect checkerboard
   end if
   go to next card
end mouseUp
```

instead of

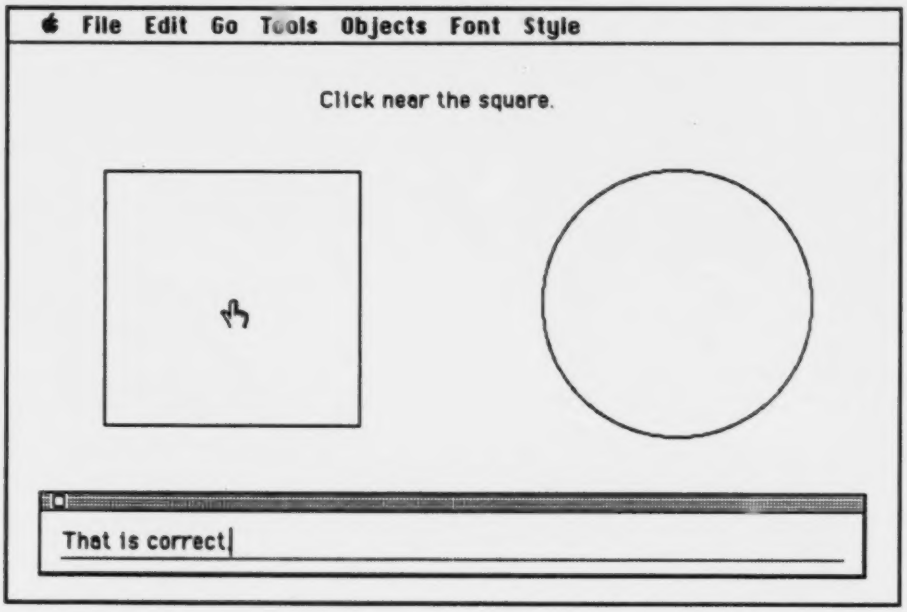
```
on mouseUp
   if random(10) < 6
   then beep
   else visual effect checkerboard
   go to next card
end mouseUp
```

If you find these many forms of **if** confusing, simply use only the multiple statement format.

Using the If Structure

The **if** structure gives the programmer a great deal of control over the path taken through the stack. Below are several examples making use of the **if** structure in a variety of contexts.

Suppose you have a card that looks like this:



You want to check to see whether the user clicks on the correct shape, and then have the user receive the appropriate reply.

Recall the **mouseH** and **mouseV** functions. **MouseV** returns the vertical (up/down) location of the mouse while **mouseH** returns the horizontal (left/right) position of the mouse. Next, try putting the following handler on the card:

```
on mouseUp
   if the mouseH < 250 then
      put "That is correct." into Message box
      wait 3 seconds
      go to next card
   else
      put "No, you clicked on the circle." into Message box
      wait 3 seconds
      put "Try again." into Message box
   end if
end mouseUp
```

Recall the **wait** command. The simplest form of its syntax is:

```
wait [for] time [seconds]
```

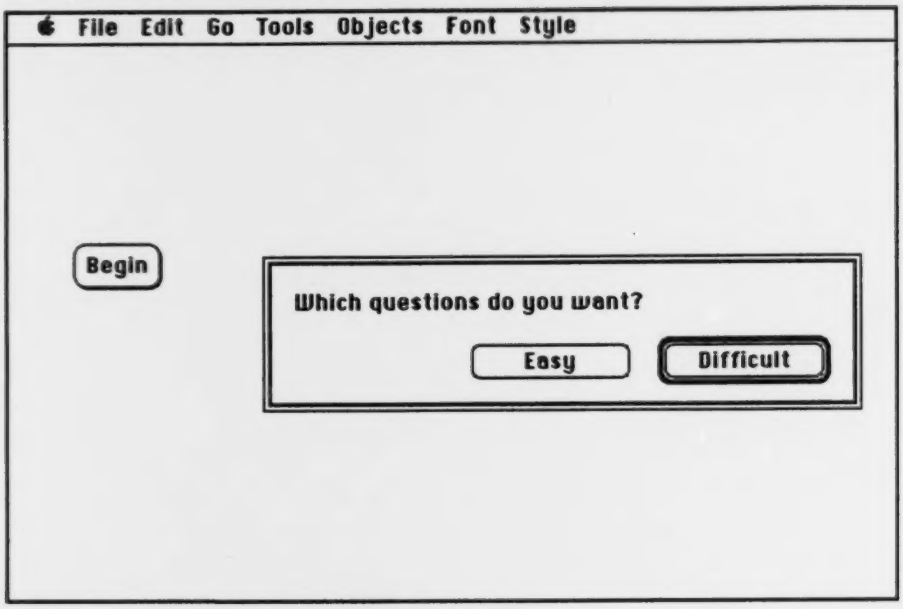
If "seconds" is omitted, *HyperCard* uses "ticks." One tick is 1/60 of a second.

If you now click on the square, you see "That is correct" in the Message box and then you see the next card in the stack. If you click on the circle, you see "No, you clicked on the circle." and then "Try again." in the Message box. Spend some time making sure you understand this handler. Did you notice that it uses the multiple statement form of **if-then-else**?

When you use **ask** and **answer**, you often want to act on the response of the user. You can use the **if** structure to test the result that is automatically put into **it** by **ask** and **answer**. For example, suppose you have a Begin button with the following script:

```
on mouseUp
   answer "Which questions do you want?" with "Easy" or¬
      "Difficult"
end mouseUp
```

If you click on the Begin button, you see



Recall that clicking on one of these buttons puts the name of that button into it. You can use if to check the contents of it:

```
on mouseUp
   answer "Which questions do you want?" with "Easy" or¬
      "Difficult"
   if it is "Easy" then
      go to card "Easy"
   else
      go to card "Difficult"
   end if
end mouseUp
```

Now *HyperCard* goes to the appropriate card based on the users' selection.

Sometimes you have more than two answers to check. Perhaps you have a menu from which to choose. Perhaps you used the form of **answer** that allows for three different choices. To check multiple responses, you can create a "case" statement. A case statement is merely a series of if statements. It looks like this:

```
if theChoice = choice 1 then action 1
if theChoice = choice 2 then action 2
if theChoice = choice 3 then action 3
if theChoice = choice 4 then action 4
   etc.
```

or

```
if theChoice = choice 1 then
   action 1
endif
if theChoice = choice 2 then
   action 2
endif
```

```
if theChoice = choice 3 then
   action 3
endif
if theChoice = choice 4 then
   action 4
endif
   etc.
```

If you expand this example to allow three choices, your script might look like this:

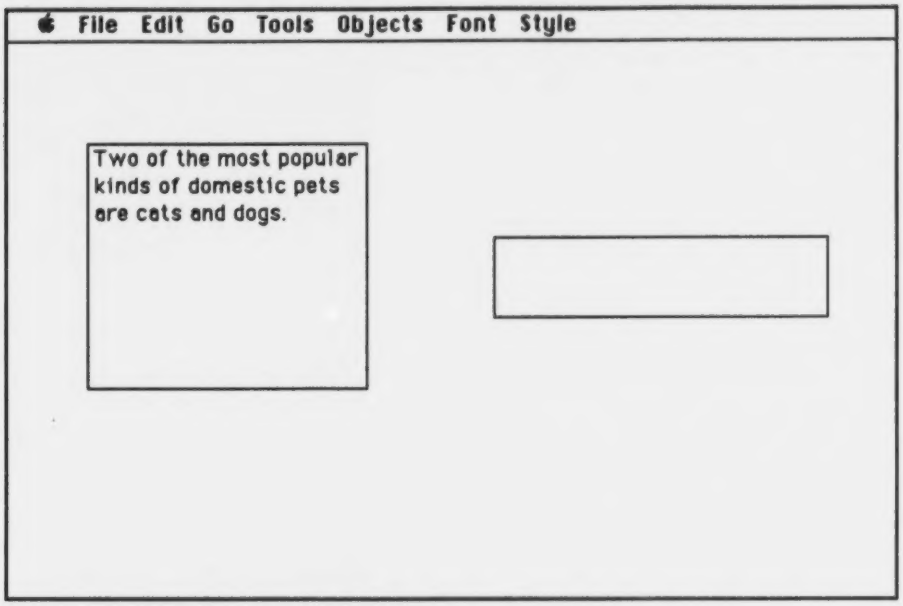
```
on mouseUp
   answer "Which questions do you want?" with "Easy" or¬
      "Difficult" or "Very Hard"
   --
   if it is "Easy" then
      go to card "Easy"
   endif
   --
   if it is "Difficult" then
      go to card "Difficult"
   end if
   --
   if it is "Very Hard" then
      go to card "Very Hard"
   end if
end mouseUp
```

Scripts with multiple **if** statements like this can be hard to read. Take time to be sure that your HyperTalk code is as clear as possible. You can always use comments to separate parts of the code, for example,

```
on mouseUp
   answer "Which questions do you want?" with "Easy" or¬
      "Difficult" or "Very Hard"
   --Beginning of case structure
   if it is "Easy" then
      go to card "Easy"
   endif
   --
   if it is "Difficult" then
      go to card "Difficult"
   end if
   --
   if it is "Very Hard" then
      go to card "Very Hard"
   end if
   --End of case structure
end mouseUp
```

In Chapter 19, you learned to use **find** and **select** to locate or select text in a field. You also learned a number of functions such as **foundChunk** and **selectedChar** to retrieve the found or selected value. You can use **if** to check the values of these functions and act on the result.

Suppose you have a card containing two fields. Put some text in the first field:



You might put this script on the card:

```
on mouseUp
   find "cats"
   if the foundText is "cats" then
      put "Cats are soft and furry." into card field¬
         "Response"
   end if
end mouseUp
```

If the user clicks over the card, a box appears around the word "cats" and the sentence "Cats are soft and furry." appears in the field to the right.

By itself, this script is not particularly useful. It looks for a specific word and then tells the user that it found that word. You can generalize this script as follows:

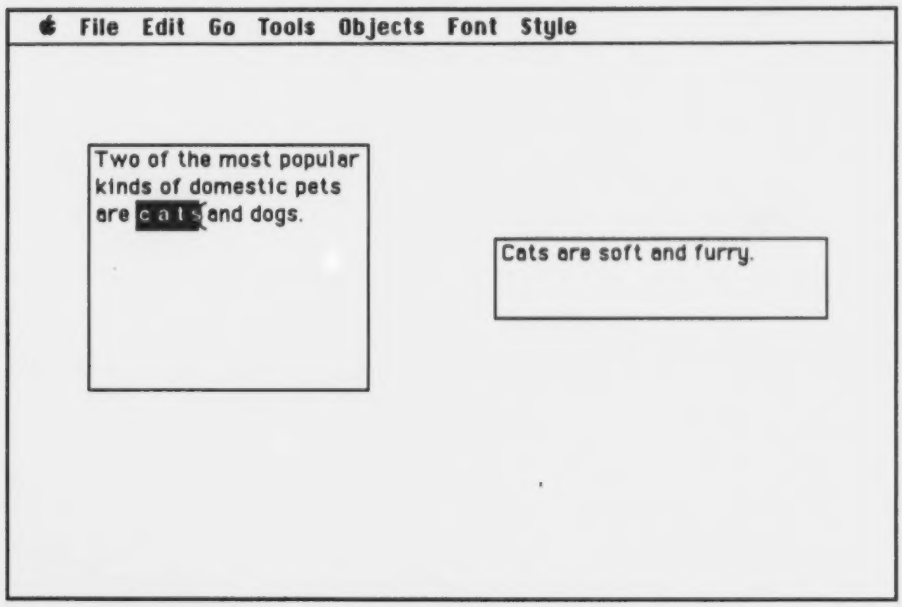
```
on mouseUp
   ask "What word do you want to find?"
   find it
   if the foundText is "cats" then
      put "Cats are soft and furry." into card field
         "Response"
   else
      put "You looked for"&&it into card field "Response"
   end if
end mouseUp
```

This is still not particularly useful. What if the user could click on any word in the text and have different responses occur depending on the word that was selected? To accomplish this, you need to use **the selection**, and make use of the message **idle**. The **idle** message is sent to the card when nothing else is happening and the Browse tool is selected. (If you are using *HyperCard* 2.0, you can show the Message Watcher box and see the **idle** messages scrolling past.)

Try putting this handler on the card:

```
on idle
   if the selection is "cats" then
      put "Cats are soft and furry." into card field¬
         "Response"
   end if
   --
   if the selection is "dogs" then
      put "Dogs are loyal and friendly." into card field¬
         "Response"
   end if
end idle
```

Now you can double-click on any word in the text. If the user double-clicks on "cats" or "dogs," messages appear in the Response card field:


File Edit Go Tools Objects Font Style
Two of the most popular kinds of domestic pets are cats and dogs.
Cats are soft and furry.

This is another technique that can be used to create hypertext.

As you can see, the **if** structure provides you with a great deal of flexibility in writing scripts. You can now have more control over the path of the user through your stacks whether you are working with numbers or words.

Debugging and Testing

There are many forms of the **if** statement given in this chapter. You really only need to remember one form:

```
if condition then
   statement list
else
   statement list
end if
```

All the other forms are derived from this one.

You can use *HyperCard*'s automatic indenting to help you check your scripts. If you type

```
on mouseUp
   if 5 = 5 then
      put "yes"
   else
      put "no"
      end mouseUp
```

you see that the **end** has not moved back in line under the **on**. *HyperCard* is waiting for you to add an **end if**. Use this feature of *HyperCard* to help detect your errors.

Keep in mind that statements that occur *after* an **if** structure run no matter what happens as a result of the condition after the **if**. That is, in

```
on mouseUp
   if random(10) < 6
   then visual effect dissolve slowly
   else visual effect slide left
   go to next card
end mouseUp
```

no matter what random number is selected, *HyperCard* will move to the next card. The visual effect used depends on the number picked. It can be difficult to see which statements are actually after the **if** structure when you use the single statement form. If you have problems with this idea, always use the multiple statement form:

```
on mouseUp
   if random(10) < 6
   then
      visual effect dissolve slowly
   else
      visual effect slide left
   end if
   go to next card
end mouseUp
```

Here the **end if** helps remind you that "go to next card" is *after* the **if** structure.

If you are using a **case** structure like the one described previously, bugs can emerge if for some reason more than one of your conditions are true. To avoid this problem, you can add an **exit** ***messageName*** to each **if** statement. **Exit** is a keyword that can be used to stop the currently executing handler:

```
if theChoice = choice 1 then action 1 exit messageName
if theChoice = choice 2 then action 2 exit messageName
if theChoice = choice 3 then action 3 exit messageName
if theChoice = choice 4 then action 4 exit messageName
   etc.
```

New HyperTalk Words in This Chapter

Functions	Messages
mouseH	idle
mouseV	

Keywords	
`if`	`else`
`then`	`exit`

Project Ideas

Spend some time working with the different forms of the **if** structure. Be sure you understand how each one works.

Expand each of the examples given in this chapter. Be sure you understand exactly what the **if** statement is doing in each one.

Make a "quiz" stack in which you check the answers given by students and provide appropriate feedback depending on their answers.

Chapter 22. Math Functions

This chapter provides a list of mathematical functions available in HyperTalk. The first part of the chapter lists commonly used functions. The second part lists more specialized math functions.

HyperCard has a wide variety of mathematical functions. Some of these functions are of use to almost any *HyperCard* user while others would be of interest only to those who use more sophisticated mathematics. This chapter simply lists these functions and gives simple examples. Some of these functions will be used in examples later in the book. The last part of the chapter will be of interest to you only if you work with more sophisticated mathematical ideas.

Recall that many of the functions in HyperTalk can be used in two forms:

```
function (factor)
```

or

```
the function of factor
```

For example, you can type

```
date ()
```

or

```
the date
```

to see the current date in the Message box. When both forms are allowed in mathematical functions, an example of each will be given.

Common Math Functions

Average

The **average** function returns the average of a list of numbers. Thus, if you type

```
average(10, 20, 30)
```

the average of the three numbers, which is 20, appears in the Message box.

Max and Min

These two functions allow you to find the largest number and the smallest number in a list. If you type

```
max(1, 3, 5, 7)
```

the number 7 appears in the Message box. Of course, you can use **put** with mathematical functions. If you type

```
put min(7, 3, 5, 2, 8)
```

the number 2 appears in the Message box.

CharToNum and NumToChar

These two functions allow you to convert between keyboard characters and ASCII values. ASCII is the numeric code used to represent keyboard characters in the

computer. Thus, for example, the character "A" has the value 65 while "B" has the value "66." All keyboard characters are represented by an ASCII value. Thus, for example, the "space" character has ASCII value 32 and the "return" character has ASCII value 13. If you type

```
charToNum("A")
```

or

```
the charToNum of "A"
```

the number 65 appears in the Message box, and if you type

```
numToChar(65)
```

or

```
the numToChar of 65
```

the letter "A" appears in the Message box.

Sqrt

This function returns the positive square root of a number. If you type

```
sqrt(16)
```

or

```
the sqrt of 16
```

the number 4 appears in the Message box. If you put a negative number into the parentheses, the result will be NAN(001), which means "not a number."

Round

The **round** function returns a number rounded to the nearest integer. Odd numbers ending in 5 round up; even integers ending in 5 round down. The following table gives examples.

`round(3.5)` becomes 3	`round(-3.5)` becomes -4
`round(3.51)` becomes 4	`round(-3.51)` becomes -4
`round(4.5)` becomes 4	`round(-4.5)` becomes -4
`round(4.51)` becomes 5	`round(-4.51)` becomes -5

Note that the rule of thumb is that if a number ends in 5 and is rounded to the next most significant digit, that number becomes even.

Trunc

The **trunc** function returns the integer part of the number passed to it. Thus

```
the trunc of 4.56
```

or

```
trunc(4.56) into
```

puts 4 into the Message box while

```
trunc(-4.56)
```

puts -4 into the Message box.

Value

The **value** function evaluates a string and returns the value of the result. For example, you might write

```
put "5 + 6 - 3" into it
```

In this example, "5 + 6 - 3" is a math problem that is put into the variable **it**. If you type

```
put it into the Message box
```

you see "5 + 6 - 3." If you type

```
put the value of it into the Message box
```

you see the number 8. **Value** evaluates the string of numbers and displays the result.

Mathematical Functions

All of the following functions can be written in the form

```
function(factor)
```

or in the form

```
the function of factor
```

Abs returns the absolute value of the number passed to it. That is, it makes the sign of the number positive.

Atan returns the trigonometric arc tangent or inverse tangent in radians. You can convert radians to degrees by multiplying by 180 and dividing the result by π.

Sin, cos, and **tan** return the sine, cosine, and tangent, respectively, of the angles given to them in radians. You can convert angles in degrees to radians by multiplying by 180 and dividing by π.

Exp returns the constant *e* (the constant 2.718...) raised to a power. Thus

```
exp(2)
```

returns *e* (2.718...) raised to the second power.

Exp1 returns **exp**(*number*) - 1.

Exp2 returns its input raised to the second power. Thus

```
exp2(3)
```

puts 8 into the Message box.

Ln returns the natural log (base *e*). **Ln1** returns **ln**(1 + *number*), while **log2** returns the logarithm to the base 2 of the number passed to it.

There is also a *constant* that is useful in mathematics. This constant, **pi**, can be used in place of writing the numeric value of the mathematical constant pi. The value of **pi** used by *HyperCard* is accurate to 20 decimal places.

New HyperTalk Words in This Chapter

Functions		
abs	exp2	round
atan	ln	sin
average	ln1	sqrt
charToNum	log2	tan
cos	max	trunc
exp	min	value
exp1	numToChar	

Constants
π (pi)

Project Ideas

Spend some time experimenting with the mathematical functions with which you are comfortable. Be sure you understand how they work. Practice using both forms of these functions.

Create a calculator card. You could create a standard calculator or you could create a scientific calculator. Be creative. How will you interact with the user? Will there be number buttons like the ones on a standard calculator, or will there be only function buttons?

Chapter 23. Containers

In your work with HyperTalk you have seen four different containers: the Message box, fields, **the selection**, and **it**. In this chapter you will examine the idea of containers in greater depth and learn to create your own containers.

In *HyperCard* a *container* is a place where you can store a value. Recall that if you type

```
the style of card button 1
```

the name of the style of the first card button will appear in the Message box. You can also use **put** to place information in the Message box. For example,

```
put "Hello there"
```

places the words "Hello there" in the Message box. The Message box is a one-line container. Whatever appears in the Message box is replaced by the next **put** command or by typing new information directly into the Message box.

Technical Note: If you copy more than one line of text onto the clipboard and then paste that text into the Message box, you see only one line. However, if you delete the first line, the second line of the pasted text appears. Note that a "line" ends with a carriage return.

The Message box is one of the kinds of containers you can use in *HyperCard*.

Fields are a second kind of container. This can be a bit confusing because a field is an object, just as buttons and cards are objects in the *HyperCard* environment. However, as you have seen previously, a field can also hold information. Recall that you can type

```
put "Hello there" into card field 1
```

and the words "Hello there" appear in the first card field, replacing any information that was previously in the field.

Fields can contain as many as 30,000 characters. Keep in mind that spaces, punctuation, and return characters are all counted as characters. If you put too many characters into a field, the extra characters are ignored and you will get an error message.

You have also seen the container **the selection**. Whenever text in a field is selected, that text is put into the container called **the selection**. You can use an **if** statement to test the value of **the selection** or you can put the value of **the selection** elsewhere.

The fourth kind of container you have seen is the container **it**. You learned that **it** automatically receives the value of the answer given to **ask** or **answer**. You will learn more about **it** in this chapter.

Variables

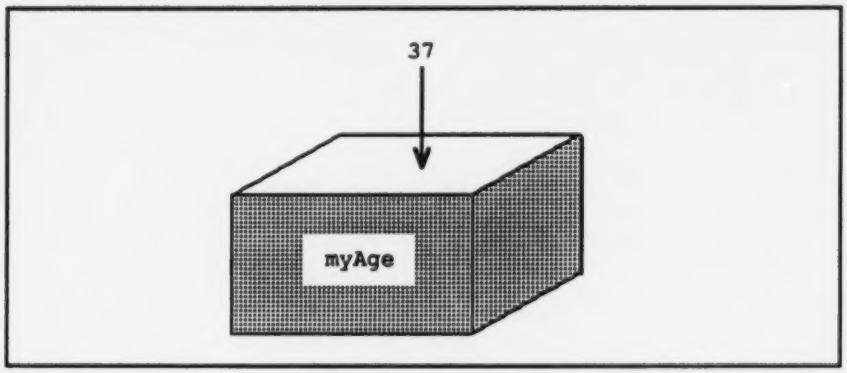
HyperCard also has containers called *variables*. A variable is a container to which you give a name. Unlike the Message box and fields, a variable is not represented visually. You can name a variable with any combination of letters and numbers. The name must begin with a letter and you can use the underscore character (_) in the name as well. Variables are "local" to the script in which they are defined and used. That is, they are only understood in the script in which they are defined.

Technical Note: Variables can be understood outside of the script where they are defined if they are declared to be "global." This concept will be discussed later in this chapter.

You give variables a value with the **put** command. Thus, you might type

```
put 37 into myAge
```

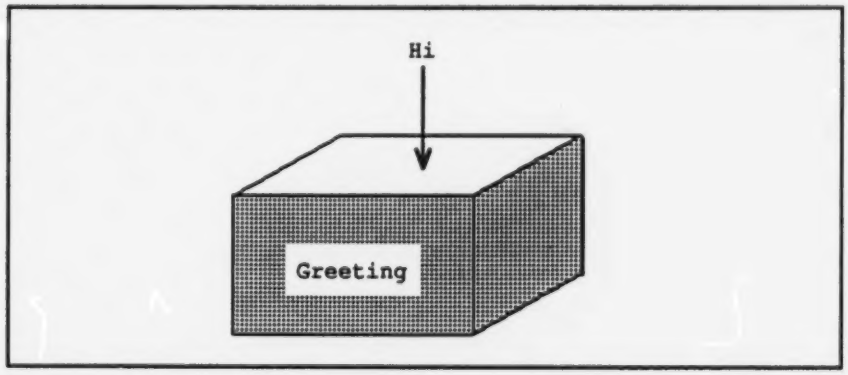
You can think of the number 37 being put into a "box" so that it can be used later:



Or you might type

```
put "Hi" into Greeting
```

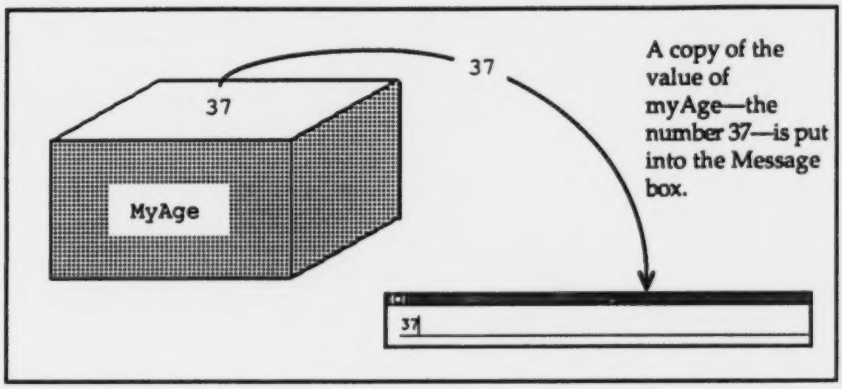
Again, you can think of the string "Hi" as being put into the Greeting "box":



In these two examples, "myAge" and "Greeting" are variables. That is, they are named containers that contain values. After typing the above commands, you can type

```
put myAge into the Message box
```

and the value of **myAge** is put into the Message box:



Or, you can type

```
put myAge into card field 1
```

and the number 37 appears in the first card field. A *copy* of the value of the variable **myAge** is put into the card field.

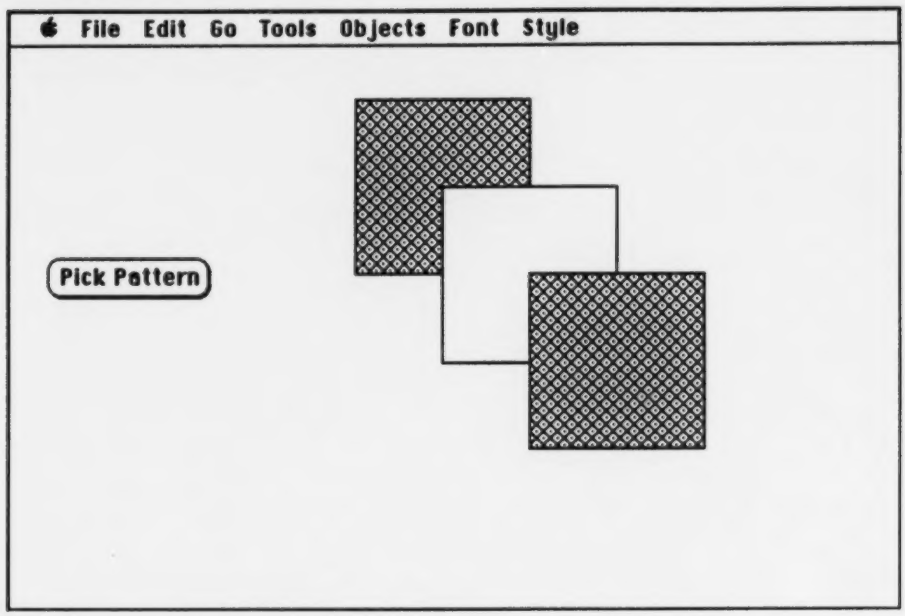
Thus "myAge" is a container holding a value that can then be put elsewhere at a later time. Notice that variables behave much like fields. However, you can see the value of variables only if you "put" them somewhere that you can look at them, while you can view the contents of fields or the Message box at any time.

Using Variables

Suppose you want to draw a series of squares on the card. You might enter this script on a "Pick Pattern" button:

```
on mouseUp
   choose rectangle tool
   set filled to true
   set pattern to 25
   drag from 200,50 to 300,150
   set pattern to 1
   drag from 250,100 to 350,200
   set pattern to 25
   drag from 300,150 to 400,250
   choose browse tool
end mouseUp
```

Clicking on your button you see



In this example, a particular pattern was selected. If you use the **random** function, you can select a random pattern. Modify the previous script as follows:

```
on mouseUp
   choose rectangle tool
   set filled to true
   set pattern to random(40)
   drag from 200,50 to 300,150
   set pattern to 1
   drag from 250,100 to 350,200
   set pattern to random(40)
   drag from 300,150 to 400,250
   choose browse tool
end mouseUp
```

Now when you click on the button, three squares appear. The first and last are filled with a pattern. Most of the time those patterns will be different. However, the original idea was to have the first and last square filled with the same pattern. The solution to this problem is to use a variable. In the following script, the variable **randomPat** is used to hold the number of the pattern selected with **random**. The value produced by **random(40)** is "put into the box" called **randomPat**. This value is then used to set the pattern for the first and third square:

```
on mouseUp
   put random(40) into randomPat
   choose rectangle tool
   set filled to true
   set pattern to randomPat
   drag from 200,50 to 300,150
   set pattern to 1
   drag from 250,100 to 350,200
   set pattern to randomPat
   drag from 300,150 to 400,250
   choose browse tool
end mouseUp
```

Using a variable allows you to "save" a value for later use.

Technical Note: There is another kind of variable called a parameter. Parameters will be discussed in Chapter 26.

More About It

You have used the predefined variable called **it**. No doubt you can now see that **it** behaves just like variables that you name. **It** holds values so that they can be used at a later time. Recall that a value is automatically put into **it** when you use **ask** and **answer**. There is another useful command, **get**, that automatically puts values into **it**. The syntax of the **get** command is:

```
get expression
```

Using this command, you can type something like

```
get the long name of card field 1
```

The value of the name of the card field is automatically put into **it**. If you now type

```
it
```

in the Message box, the long name of the card field appears.

By using **it**, you can rewrite the rectangle-drawing handler from the previous section:

```
on mouseUp
   get random(40)
   choose rectangle tool
   set filled to true
   set pattern to it
   drag from 200,50 to 300,150
   set pattern to 1
   drag from 250,100 to 350,200
   set pattern to it
   drag from 300,150 to 400,250
   choose browse tool
end mouseUp
```

This script functions in exactly the same way that the variable **randomPat** functions. In this case, however, the script takes advantage of the fact that **it** is always predefined. That is, you can think of the **it** "box" as always being available. You did not have to make up your own name for the variable and thus create a new "box."

Recall that **it** is *local* to the handler in which it is used. That is, it functions like the other variables you have learned about in this chapter.

Using Global

When you create a variable in a handler, that variable is local to the particular handler in which it is created. That is, its value is available only when that script is running and is destroyed when the handler ends. What happens, then, when you want to break a script into small, more readable parts? Suppose you decide to modularize the earlier version of the rectangle-drawing example. You might write

```
on mouseUp
   pickRandom
   setUp
   firstRect
   secondRect
   thirdRect
   resetTool
end mouseUp

on pickRandom
   put random(40) into randomPat
end pickRandom

on setUp
   choose rectangle tool
   set filled to true
end setUp

on firstRect
   set pattern to randomPat
   drag from 200,50 to 300,150
end firstRect

on secondRect
   set pattern to 1
   drag from 250,100 to 350,200
end secondRect

on thirdRect
   set pattern to randomPat
   drag from 300,150 to 400,250
end thirdRect

on resetTool
   choose browse tool
end resetTool
```

If you try to run these handlers, you will get errors. **FirstRect** does not have access to the value of **randomPat** that was created in **pickRandom**. The solution to this problem is the **global** keyword. You declare the variable to be global in every procedure that uses it:

```
on mouseUp
   put empty
   pickRandom
   setUp
   firstRect
   secondRect
   thirdRect
   resetTool
end mouseUp
```

```
on pickRandom
   global  randomPat
   put random(40) into randomPat
end pickRandom

on setUp
   choose rectangle tool
   set filled to true
end setUp

on firstRect
   global  randomPat
   set pattern to randomPat
   drag from 200,50 to 300,150
end firstRect

on secondRect
   set pattern to 1
   drag from 250,100 to 350,200
end secondRect

on thirdRect
   global  randomPat
   set pattern to randomPat
   drag from 300,150 to 400,250
end thirdRect

on resetTool
   choose browse tool
end resetTool
```

Now the script will run as you expected. Note that you cannot declare it to be global. It will always be local, so if you need to use a variable in several procedures, you cannot use it.

Debugging and Testing

Be careful that you don't confuse **me** and **it**. **It** is a predefined container in *HyperCard*. Values are automatically put into **it** from a number of *HyperCard* commands. For example, if you type

```
get the id of card button 1
```

the **id** number of the first card button automatically goes into **it**. You can then use that value elsewhere in scripts on any *HyperCard* object. Keep in mind that **it** can be accessed from any script and values can be put into **it** from any script.

Me, however, is a special object descriptor. **Me** is *not* a container. It does not hold a value. It is used to refer to the current object. Thus, you might type

```
on mouseUp
   get the id of me
end mouseUp
```

The **id** of the current object would go into **it**. Take special note that the value of **it** after execution of this handler depends on the object in which the handler is placed. Putting this handler on the first card button would give an entirely different result than putting it on the third card field.

It is easy to forget to put **global** in each handler that uses a particular variable. If you are getting errors in handlers using variables, check to be sure all of the variables are declared to be global in every procedure that uses them.

Keep in mind that you cannot declare it to be global. HyperTalk will accept the command

```
global it
```

However, if you try to access the value of it in several handlers, you will not get the results you expect.

At a number of places in this chapter, you have been reminded that variables and it are local to the handler in which they are used. However, you can declare a variable in the Message box and use that variable again from the Message box. Similarly, you can put values into it and use them in the Message box. However, these values will not be available from handlers.

New HyperTalk Words in This Chapter

Commands	Keywords
get	global

Project Ideas

Spend some time experimenting with variables. Put values into variables. Put the values into the Message box. Put the values into fields. Experiment until you are comfortable with this new idea.

Have you tried to write scripts before that really needed variables? If so, try rewriting some of them, using this new idea.

If the idea of variables is new to you, spend some time doing some simple experiments. Type commands like

```
put the name of this card into cardName
put cardName
```

or

```
get the name of this card
put it
```

to help you begin to understand how variables get values and how to use those values.

Chapter 24. Looping

In Chapter 15 you learned about the **repeat** structure. You saw that you could use this structure to do a series of actions a specific number of times. In this chapter you will learn a number of other ways to use **repeat**. You will also learn to use a form of **wait** that can be used to create a loop.

In Chapter 21 you learned about branching. You learned that you could control the path taken through your stack based on the result of some condition. Branching is an important component of any programming language. Another component of all programming languages is *looping*. Looping allows a programming language to repeat a task over and over. You have seen that in HyperTalk you can create a loop by using the keyword **repeat**.

Repeat For

In Chapter 15 you used the **repeat** structure to create simple animation. To accomplish this, you used the **repeat for** form of **repeat**:

```
repeat [for] integer [times]
   statements list
end repeat
```

Suppose you have a sequence consisting of three cards containing slightly different pictures. When users click on a button, you want them to see the animation. Your script might look like this:

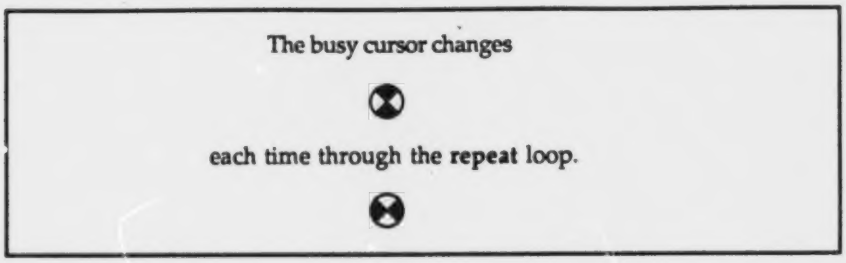
```
on mouseUp
   repeat for 3 times
      go to next card
   end repeat
end mouseUp
```

The user will see the three cards flip past. If the animation moves too quickly, you can insert a **wait** command, for example,

```
on mouseUp
   repeat for 3 times
      go to next card
      wait 15 ticks
   end repeat
end mouseUp
```

Another interesting use of the **repeat for** form causes the "busy" pointer to appear to spin. Each time **cursor** is set to "busy," the image turns 45 degrees clockwise. Thus, you can write

```
on mouseUp
   repeat for 20 times
      set cursor to busy
      wait 2 ticks
   end repeat
end mouseUp
```



This can be used to create a "pause" in some activity occurring in your stack, for example,

```
on mouseUp
   repeat for 20 times
      set cursor to busy
      wait 2 ticks
      doAction --Some other action occurs here
   end repeat
end mouseUp
```

Repeat With

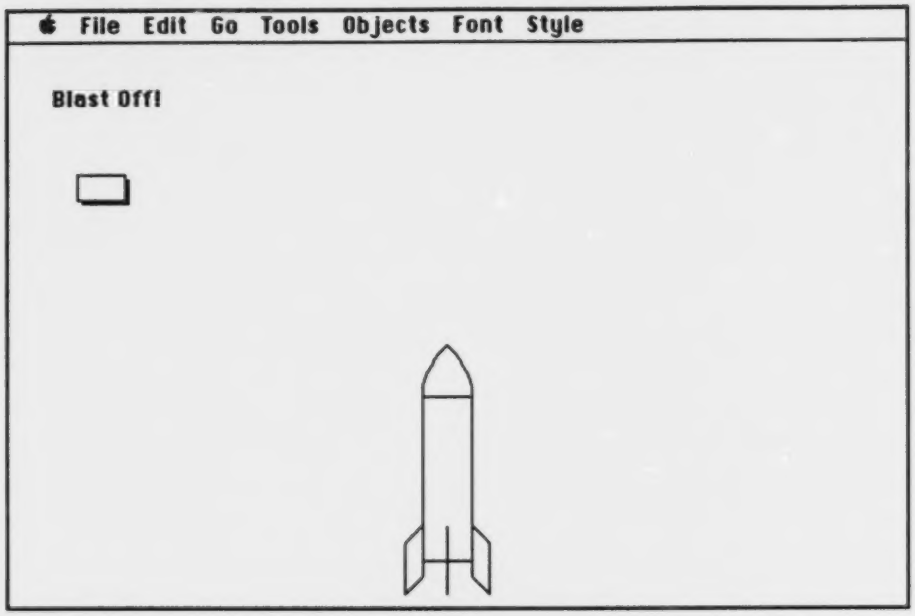
There is also a form of the **repeat** structure that uses a variable to loop a specific number of times. This form is similar to the FOR-NEXT statement in BASIC and a number of other programming languages. **Repeat with** has two forms

```
repeat with variableName = integer1 to integer2
   statements list
end repeat
```

```
repeat with variableName = integer2 down to integer1
   statements list
end repeat
```

Note that "integer1" must be less than or equal to "integer2" for both forms. Also note that the first form is used to count up and the second is used to count down.

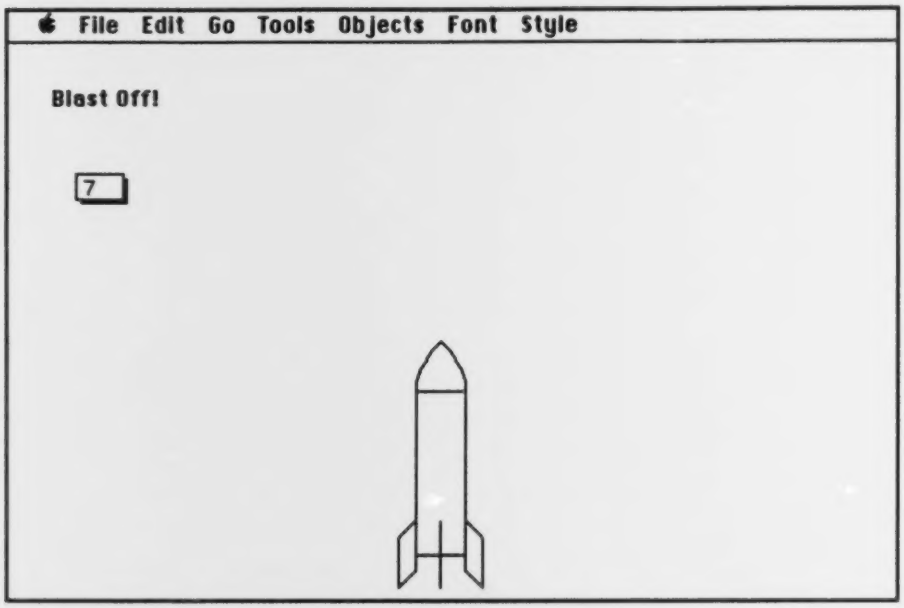
Suppose you have a picture of a rocket ready to blast off:



You could use the **repeat with** form of the **repeat** statement to display a countdown before the rocket takes off. Because the rocket countdown is to go backwards, you to need use the second form of the **repeat with**:

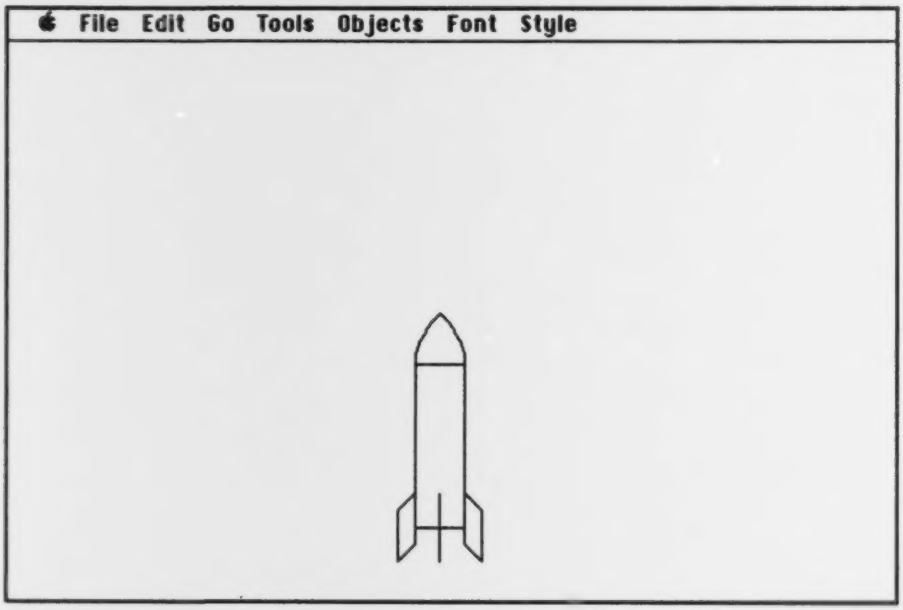
```
on mouseUp
   repeat with counter = 10 down to 1
      put counter into card field "Count down"
      wait 1 second
   end repeat
end mouseUp
```

Clicking on the Blast Off! button causes the numbers from 10 to 1 to appear one at a time in the field:



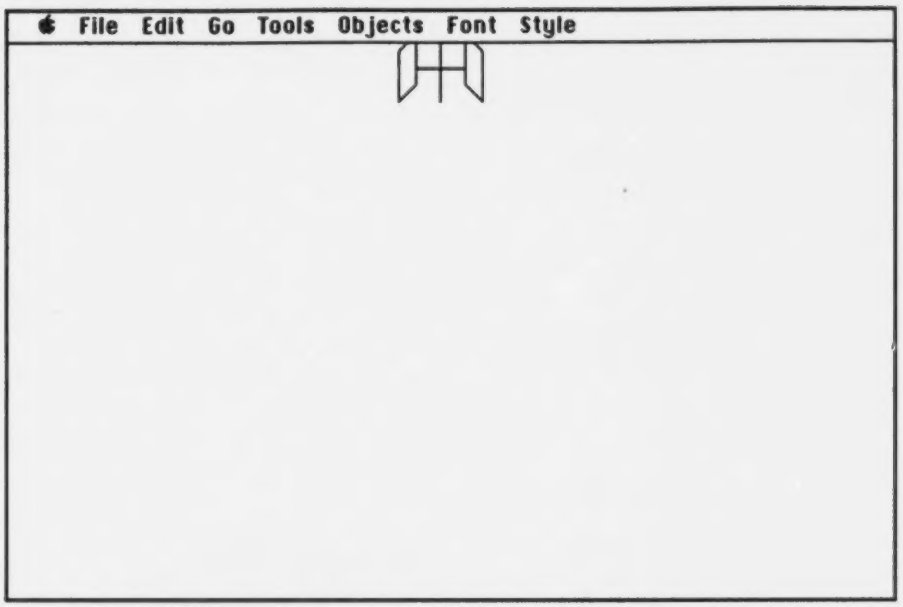
Repeat While and Repeat Until

Next, you might want to animate the rocket so that it moves off the card. You could use drag to move the rocket, or you could create a series of cards that show the rocket moving up and off the card. For example, the previous card shown might be named "Frame 1." The next card could then be called "Frame 2":



250

The final card could be called "Last Frame":



Now you want to flip through the cards until the Last Frame card is reached.

There are two other forms of **repeat** structure you might use to show this animation:

```
repeat until condition
   statement list
end repeat
```

or

```
repeat while condition
   statement list
end repeat
```

Here, "condition" has the same meaning it had when you were working with **if**. You might add the following code at the end of the **mouseUp** handler on your Blast Off! button:

```
repeat until the short name of this card is "Last Frame"
   go to next card
   wait 30 ticks
end repeat
```

Now when you click on Blast Off! you see a countdown and then see the rocket move up the screen.

Note that you could also write the above code as

```
repeat while the short name of this card is not "Last Frame"
   go to next card
   wait 30 ticks
end repeat
```

Sometimes one of these two forms seems more natural or English-like than the other. Sometimes one form is easier to use than the other. Don't always use the same form just because you like it—think carefully before writing your **repeat** scripts.

Modularizing Your Scripts

As your scripts get longer and more complex, it becomes increasingly important to take time to break them into smaller pieces. Your code is then more readable and easier to debug. Currently the handler on Blast Off! looks like this:

```
on mouseUp
   repeat with counter = 10 down to 1
      put counter into card field 1
      wait 1 second
   end repeat
   repeat until the short name of this card is "Last Frame"
      go to next card
      wait 30 ticks
   end repeat
end mouseUp
```

You could rewrite it as follows

```
on mouseUp
   countDown
   blastOff
end mouseUp

on countDown --displays a counter
   repeat with counter = 10 down to 1
      put counter into card field 1
      wait 1 second
   end repeat
end countDown

on blastOff --animates the rocket
   repeat until the short name of this card is "Last Frame"
      go to next card
      wait 30 ticks
   end repeat
end blastOff
```

It is much easier for you and for other people to understand your code if it is broken into smaller parts.

Wait Loops

As indicated at the beginning of this chapter, there are two forms of the **wait** command that are really special loops. Their syntax is:

```
wait until condition
wait while condition
```

The *conditions* in these two forms of **wait** work exactly like the similar conditions in **repeat until** and **repeat while**. The **wait** loops are really special cases of the corresponding **repeat** loops. The **wait** loops can only pause—no other action can take place while a **wait** loop is running. Here are a couple of examples of the use of **wait**.

```
wait until the mouseH > 300
```

waits until the user moves the mouse into the right-hand part of the card. If you put this statement in the Message box and press Return, you see the I-beam. Slowly move

the mouse to the right across the card. When you pass the horizontal coordinate of 300, the mouse pointer will change to the Browse tool.

The function **mouse** checks to see if the mouse button is up. Its syntax is:

```
the mouse
mouse()
```

For example,

```
wait while the mouse is up
```

causes *HyperCard* to pause until the mouse button is clicked.

*Technical Note: In HyperTalk, **up** and **down** are constants that are equivalent to **true** and **false**.*

Try typing this statement in the Message box. After you press Return, the mouse pointer is the I-beam until you click the mouse button. A similar function,

```
the mouseClick
mouseClick()
```

returns **false** if the mouse button is not **down**. When you use **mouseClick**, *HyperCard* waits until the mouse button is released, and then returns **true**. Thus, you can also type

```
wait until the mouseClick is true
```

to pause until the mouse is clicked.

Debugging and Testing

Have you ever "lost" buttons or fields on your card or background? You can use **repeat** to help find them. For example, you might write

```
on mouseUp
   repeat with num = 1 to the number of card buttons
      show card button num
   end repeat
end mouseUp
```

A similar script could be written for finding lost fields.

New HyperTalk Words in This Chapter

Commands
`wait while`
`wait until`

Functions
`mouse`
`mouseClick`

Constants
down
up

Keywords
repeat

Project Ideas

Spend some time working through the examples given in this chapter. Enhance them. Make sure you understand how each form of **repeat** works.

Create an animation using several cards. Try using different forms of the **repeat** command to display your animation.

Create a button-retriever script based on the ideas in the Debugging and Testing section in this chapter. Perhaps you could have the script automatically move any buttons that are off the side of the card back onto the card. Or perhaps you could have the script notify the user as each button off the screen is located. Then ask the user if the button should be moved back onto the card.

Chapter 25. Using Parameters

You have learned how to create variable names and use them to store values. When declared to be global, these variables can be used to access values in a variety of different handlers. In this chapter you will learn a different way to move values from one handler to another.

Earlier you learned a great deal about variables. You learned that

- you can create variables whenever you need them
- you can use almost any name you want for a variable
- values of the variables can be used anywhere in the handler in which they are defined
- without **global**, variables are *local* to the handler in which they are used
- you must use **global** to make use of a variable in more than one handler
- **global** must be in every handler in which you use a global variable
- you can use global variables to pass values from one object to another
- HyperTalk has a built-in variable named **it**
- **it** is always local.

In this chapter you will learn about parameters. There are many similarities between variables and parameters. You can use any name you want for parameters. Parameters are used to hold values. However, parameters are *local* to the handler in which they are defined and yet they can be used to pass values among objects.

You have used HyperTalk words that require parameters, for example, the command **hide**. If you just type

```
hide
```

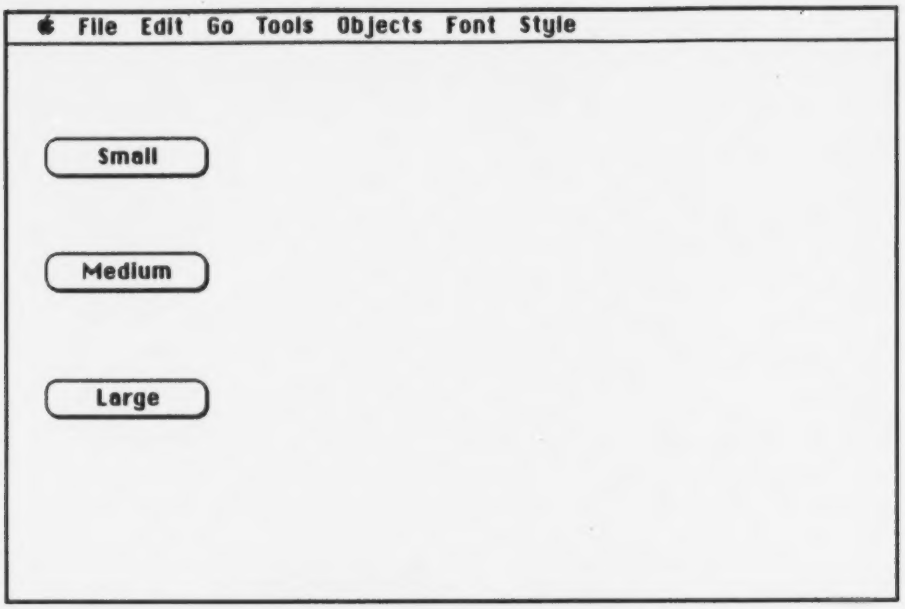
in the Message box, you get an error message. Similarly, typing the function

```
the number
```

produces an error message. **Number** and **hide** require inputs. You can also say that these two HyperTalk words have parameters for which you must give values. Can you think of other HyperTalk commands or functions that require parameters?

Getting Started With Parameters

Suppose you want to draw circles of three different sizes using different buttons. You might have a card that looks like this:



To accomplish this task, you could put the following script on the Small button:

```
on mouseUp
   choose oval tool
   set centered to true
   drag from 320,150 to 320 + 50, 150 + 50
   choose browse tool
end mouseUp
```

Then you would need to put a similar script on the Medium button:

```
on mouseUp
   choose oval tool
   set centered to true
   drag from 320,150 to 320 + 75, 150 + 75
   choose browse tool
end mouseUp
```

Finally, you would put yet another similar script on the Large button:

```
on mouseUp
   choose oval tool
   set centered to true
   drag from 320,150 to 320 + 100, 150 + 100
   choose browse tool
end mouseUp
```

Look carefully at these handlers. In what way are they different? Notice that the three handlers are written is such a way as to emphasize their similarities. The only difference among these handlers is the number added to the second set of coordinates in the **drag** command. When you have very similar procedures that differ only slightly from each other, it is likely that a parameter will be useful.

Begin by creating a card similar to the one shown previously. In the script editor of the *card*, put

```
on drawCirc circSize
   choose oval tool
   set centered to true
   drag from 320,150 to 320 + circSize, 150 + circSize
   choose browse tool
end drawCirc
```

Compare this handler with the previous three handlers. How is this handler similar to those handlers?

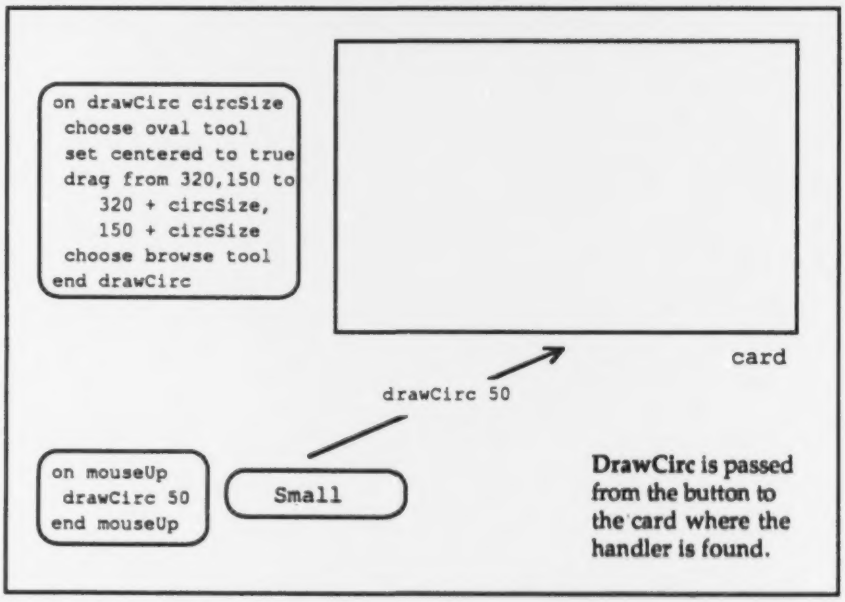
Did you note that the value added to 320 and 150 in the previous three handlers was replaced by **circSize**? Did you also note that **circSize** was added to the first line of the handler?

CircSize is a parameter. It is similar to a variable. It holds a value that can be used later in the handler. In this case, if the handler receives the number 100 as input, 100 is added to both 320 and 150 and a large circle is drawn. If the handler receives the number 50, then 50 is added to both 320 and 150 and a small circle is drawn.

Since you have your **drawCirc** handler on the card, you can put scripts on the buttons that make use of **drawCirc**. First, put the script

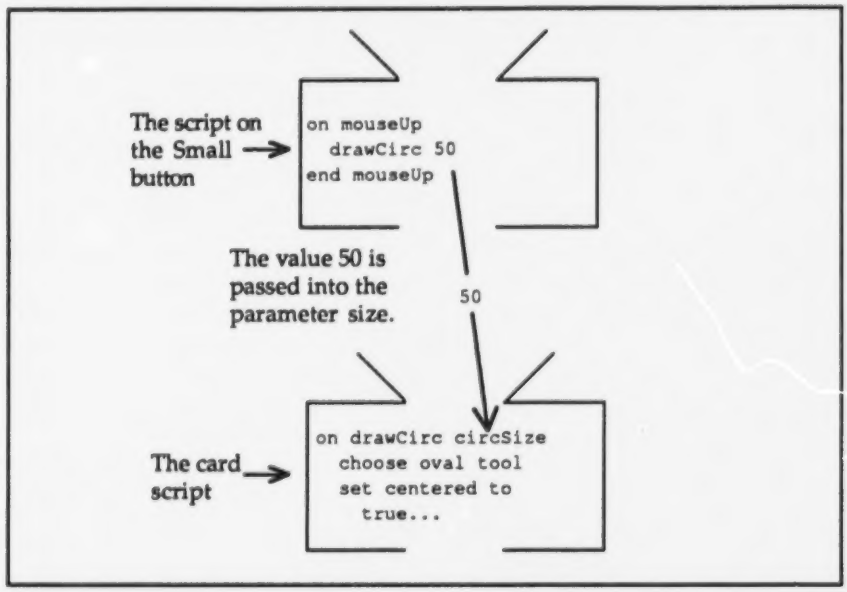
```
on mouseUp
   drawCirc 50
end mouseUp
```

on the Small button. When you click on the button, a small circle is drawn. Remember that when *HyperCard* encounters **drawCirc** and it does not find such a handler on the button, it moves along the message-passing hierarchy until it locates such a handler. In this case, it finds the appropriate handler on the card:



(Note: As you work on this example, you may want to add a button to clear the graphics from the card after each circle is drawn.)

In addition to the **drawCirc** message, the number 50 is sent from the button to the card. In the handler on the Small button, the number 50 is used with the handler **drawCirc**. This value, along with the **drawCirc** message is sent along the message-passing hierarchy. We say that 50 is "passed" to the **drawCirc** handler. Here is another way to visualize what is happening:



The **drawCirc** message is passed along the message-passing hierarchy until it encounters the **drawCirc** handler on the card. The number 50 is then given to **drawCirc**. Then the number 50 is given to the local parameter—container—**circSize**, and then the appropriate size of circle is drawn.

Note that the value of **circSize** is not available to any other handler you might put on the card. It is understood only in the handler **drawCirc**. This is the difference between parameters and local variables. You have seen that you must use **global** to use a variable in more than one handler. Local variables cannot be used to pass information from one object to another—or even from one handler to another on the same object. However, *parameters* can be used to pass a value from one *HyperCard* object to another without making the name global to your entire stack.

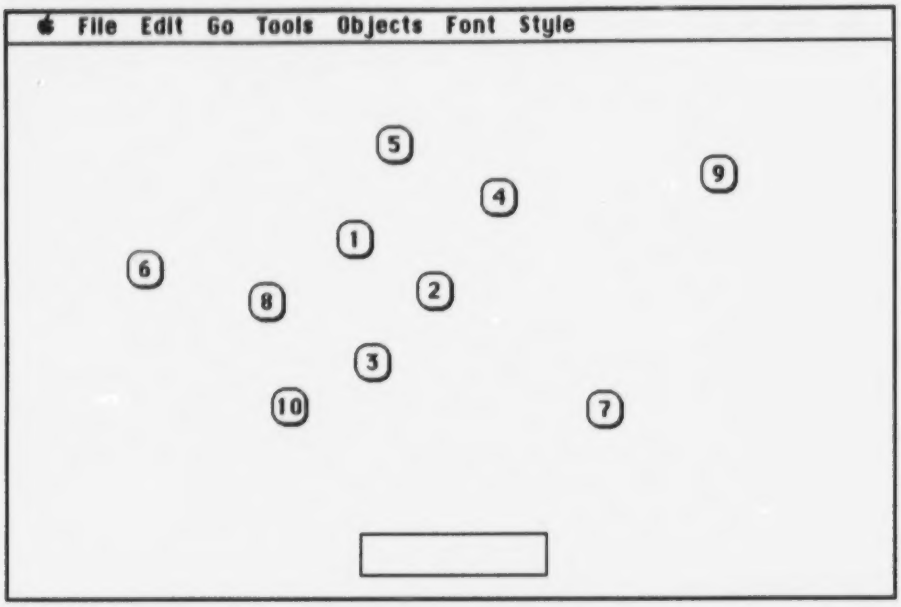
Parameter passing is a fairly sophisticated idea. If your only programming experience is with HyperTalk or an older version of BASIC, then this is a complete new concept for you. Give yourself time. Don't expect to completely understand what is happening immediately. As your develop your HyperTalk programming skills, you will begin to see where parameter passing would be of value.

More Than One Parameter

After you are comfortable with the **drawCirc** example, you should work through this next example to better understand parameter passing. However, if parameter passing is a completely new idea, you may want to skip this section.

To explore working with multiple parameters, create a sort of arcade game. There will be 10 "balls." If you touch the ball, it will scoot away. If you click on the ball, you

win the game. Begin by creating "round rectangle" buttons and resizing them so that they look like little balls:

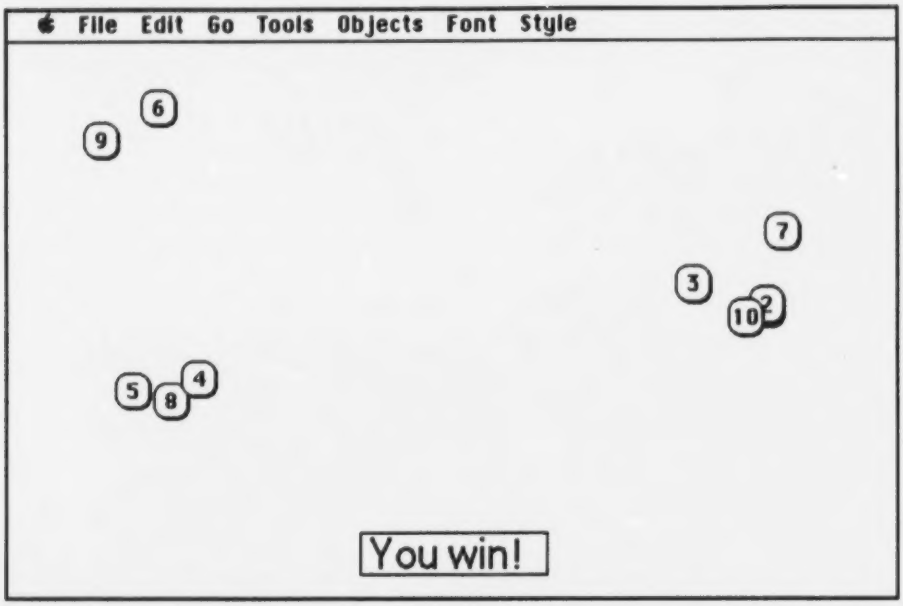


Name each of these ball buttons with a number. Each button contains the following script:

```
on mouseUp
   beep 1
   put "You win!" into the Message box
end mouseUp

on mouseWithin
   moveButton (the short name of me), (20 * the value of¬
      the short name of me)
end mouseWithin
```

If the button receives a **mouseUp** message, you hear a beep and "You win!" appears in the Message box:



However, whenever the button receives a **mouseWithin** message, it sends a **moveButton** message. The **moveButton** handler is on the card:

```
on moveButton buttonToMove, amountToMove
   put random(400) into xdistance
   put random(200) into ydistance
   set the location of card button buttonToMove to ¬
      amountToMove + xdistance, amountToMove + ydistance
end moveButton
```

Note that **moveButton** has two parameters, **buttonToMove** and **amountToMove**. Each button sends the value "the short name of me" to the parameter **buttonToMove**. This is used to identify the correct button to move. The second parameter is used to determine how far to move the button. The larger the number that names a button, the farther the button is likely to move. If this game were developed further, this technique could be used to make higher numbered balls harder to catch. The **moveButton** handler then picks two random numbers and adds those to the value of **amount** to determine where to move the specific button.

Debugging and Testing

Earlier in this chapter, you read that parameters can be used to pass values among *HyperCard* objects. You can also declare global variables to move values from one object to another. While using **global** will make variable names accessible anywhere in the stack, using global variables can cause "side effects." That is, it is easy to lose track of the names you used in various scripts. If variables are declared globally, you may accidentally use a variable name twice, causing mysterious bugs to appear. As your scripts become more complex, it is increasingly important that you use parameter passing whenever possible to pass values among objects.

Project Ideas

Extend the first example given in this chapter so that you can pass both the size and the location of the circle. That is, you might have

```
on drawCirc size, xcor, ycor
```

as the first line of your handler.

Experiment with the second example given in this chapter. Can you catch a ball? Think carefully about what message passing is going on as you play this game. How can you make the game more interesting?

Create some "tool" handlers that draw simple shapes. Use parameters to pass values such as the pattern number, whether the shape is filled, the size of the line, and so forth. Then create buttons that pass random values to some of your "tools." Can you create a card that allows the user to click on a variety of buttons to create random modern art?

Chapter 26. Where Do I Go From Here?

The introduction mentioned that this book contained three general sections. The first few chapters were primarily review and assumed that the reader had experience using *HyperCard* and merely needed to refresh his or her memory. The second part of the book, Chapter 3 through Chapter 20, covered HyperTalk programming and new HyperTalk words and how to use them. It introduced computer science ideas such as programming style and the details of syntax of the commands and functions. The last part of this book focused on programming concepts that are important in most programming languages.

If you have had experience programming in other languages, you no doubt had little difficulty with these last few chapters. In your other programming experience, you encountered conditionals, loops, parameters, and the like. The more programming experience you have had, the easier these last few chapters have seemed.

However, if HyperTalk is your first programming language, you probably found the last few chapters rather overwhelming. Indeed, many powerful and important ideas are covered rather briefly and in a short space. Don't be concerned if you feel a bit overwhelmed at this point.

So, how do you proceed from here in your work with *HyperCard*? Whether you are a novice or experienced programmer, you should add some other *HyperCard* books to your library and begin to make use of them. If you are using version 1.2.5, you should consider Apple's *HyperCard Script Language Guide* (Addison-Wesley) and if you are using version 2.0, you should make use of *HyperTalk—The Book* by Winkler and Kamins (Bantam). In these books, you will find other features of HyperTalk not discussed in this book. Experiment with some of these unfamiliar commands, functions, or features. Incorporate them into your stacks.

You might also consider acquiring some public domain *HyperCard* stacks. There are many available through catalogs, through users groups, and on networks. Examine stacks that interest you. Look at the HyperTalk code and see how it works. Try modifying some of these stacks or adapting into your own work some of the ideas you see.

If you are new to programming, studying examples in other books and examining stacks others have written will help you understand the importance of ideas such as the use of conditionals and loops. These other sources will help you see the reasons for local and global variables. They will give you ideas for the use of parameter passing.

If you are an experienced programmer, looking both at reference books and at the work of other programmers will help you see how HyperTalk is different from other programming languages you know. It will help you understand the ways in which HyperTalk and *HyperCard* are different from other programming environments.

Whatever your background or experience, just reading through this book is not enough to master HyperTalk. Learning to be a good programmer means hours and hours of practice, both at writing code and at debugging.

Keep working at it—and have fun!

Appendix 1. Keyboard Shortcuts

Navigation Between Cards	
Tilda (~/ `) key	go back
Command-1	go to the first card of current stack
Command-2	go to previous card
Command-3	go to next card
Command-4	go to the last card of current stack
Command-H	go to the Home stack
Command-R	show the most recent cards

Editing	
Command-X	cut
Command-C	copy
Command-V	paste
Command-Z	undo
Command-B	toggles edit background

Working With Objects	
Command-N	create a new card
Command-Option drag	copy a selected button or field
Command- +	bring an object closer
Command- -	send an object farther away
Command-drag	create a new button or field when the appropriate tool is selected
Command-P	print a card

Files	
Command-O	open a stack
Command-Q	quit *HyperCard*

Selecting Tools	
Command-Tab	Choose Browse tool
Command-Tab-Tab	Choose Button tool
Command-Tab-Tab-Tab	Choose Field tool

Show/Hide	
Command-space	show/hide menu bar
Command-M	show/hide Message box
Command-Option	display buttons
Shift-Command-Option	display buttons and fields

Working With Scripts	
Command-. (period)	stops a script from running
Command-Option-click	goes to the script editor of a button or field when the Button or Field tool is selected
Command-Option-click	goes to the script editor of a button when the Browse tool is selected
Shift-Command-Option-click	goes to the script editor of a field when the Browse tool is selected
Command-Option-C	goes to the script editor of current card
Command-Option-B	goes to the script editor current background
Command-Option-S	goes to the script editor of current stack

Menu Bar Commands (with a Paint tool selected)	
Command-S	select graphics on the screen
Command-A	select all
Command-K	keep
Command-T	text style

Double Click on a Tool	
Select tool	selects the full screen
Lasso tool	selects all the graphics on the screen
Pencil tool	sets fatbits mode on/off
Brush tool	displays brush shapes window
Eraser tool	clears all graphics on the screen
Line tool	displays line sizes window
Rectangle	sets draw filled mode on/off
Round rectangle	sets draw filled mode on/off
Curve	sets draw filled mode on/off
Oval	sets draw filled mode on/off
Polygon tool	sets draw filled mode on/off (but will not draw filled)
Spray can	shows/hides patterns window
Bucket tool	shows/hides patterns window
Text tool	displays text styles window
Regular polygon	displays polygon sides window

Shift and Drag	
Selected graphic	can move the graphic horizontally/vertically
Pencil tool	constrains to horizontal or vertical lines
Brush tool	constrains to horizontal or vertical strokes
Eraser tool	constrains to horizontal or vertical erasures
Line tool	constrains lines to 15 degree intervals
Spray tool	constrains to spraying horizontally or vertically

Command With Click and Drag	
Select tool	selects only a rectangle surrounding a graphic
Pencil tool	fatbits mode on/off
Lasso tool	select all
Brush tool	brush uses white pattern
Spray tool	spray uses white pattern

Option With Click and Drag	
On a selected graphic	copy that graphic
On Select tool	same as Lasso tool

Command and Shift With Click and Drag	
Spray tool	spray horizontally/vertically in white pattern

Shift and Option With Click and Drag	
selected graphic	copies and moves horizontally/vertically

Command and Option With Click and Drag	
Pencil tool, fatbits mode	moves the window around

Delete Key	
Selected graphic	removes the graphic

Appendix 2. HyperTalk Words by Category

Keywords

end

```
end messageName
```

exit

```
exit messageName
exit to HyperCard
```

global

```
global variableList
```

if

```
if condition then statement [else statement]
```

```
if condition
then statement
[else statement]
```

```
if condition then
   statement list
[else
   statement list]
end if
```

on

```
on messageName [parameterList]
```

```
repeat
   repeat controlForm
      statementList
   end repeat
```

takes the following forms:

```
   repeat [forever]
   repeat [for] number [times]
      statementList
   end repeat
   repeat [for] integer [times]
      statements list
   end repeat
   repeat with variableName = integer1 to integer2
      statements list
   end repeat
   repeat with variableName = integer2 down to integer1
      statements list
   end repeat
   repeat until condition
      statement list
   end repeat
   repeat while condition
      statement list
   end repeat
```

```
send
   send "messageName [parameterList]" [to object]
```

Commands

```
answer string1 [with string2 [or string3] or string4]]]
```

```
ask string1 [with string2]
```

```
beep [number]
```

```
choose [the] toolName tool  (in HyperCard 2.0)
```

```
choose toolName tool
```

```
click at integer1, integer2 [with key1 [,key2[,key3]]]
```

```
delete chunk [of container]
```

```
doMenu menuItem
```

```
drag from integer1, integer2 to integer3, integer4 [with
   key [,key2[,key3]]]
```

```
find chars expression [in field fieldDescriptor]
find expression [in field fieldDescriptor]
find word expression [in field fieldDescriptor]
```

```
get expression
```

go [to] [stack] *stackName* **go** [to] *bkgndDescriptor* [of [stack] *stackName*] **go** [to] *cardDescriptor* **go** [to] *cardDescriptor* [of *bkgndDescriptor*] [of [stack] *stackName*]
hide groups **hide** [the] *windowName* **hide** menuBar **hide** *objectName*
play stop **play** "string1" [tempo *number*] "*string2*"
print card **print** *cardDescriptor* **print** *expression* card[s]
put *expression* [*preposition container*]
reset paint
select [*preposition*] *chunkExpression* of *fieldDescriptor* **select** [*preposition*] text of *fieldDescriptor* **select** empty **select** *objectDescriptor*
set [the] *property* [of *object*] to *string*
show [the] *windowName* [at *integer1, integer2*] **show** all cards **show** groups **show** *integer* cards **show** menuBar **show** *objectName* [at *integer1, integer2*]
visual [effect] *effectName* [*speed*] [to *image*]
wait [for] *integer* [seconds] **wait** [for] *time* [seconds] **wait** until *condition* **wait** while *condition*

Functions

The functions represented below are shown in the

```
the function of factor
```

form. They may also be written as

```
function(factor)
```

Functions preceded by an asterisk (*) may be written only in the form shown.

*the [*adjective*] **date**	the **number** of *chunkType* in *string*
the [*adjective*] **time**	the **number** of *objects*
the **clicktext**	***offset**(*string1*, *string2*)
the **foundChunk**	the **random** of *integer*
the **foundField**	the **seconds**
the **foundLine**	the **selectedChunk**
the **foundText**	the **selectedField**
the **mouse**	the **selectedLine**
the **mouseClick**	the **selectedText**
the **mouseH**	the **ticks**
the **mouseV**	

Mathematical Functions

the **abs** of (*factor*)	the **log2** of (*factor*)
the **atan** of (*factor*)	the **max** of (*factor*)
the **average** of (*factor*)	the **min** of (*factor*)
the **charToNum** of (*factor*)	the **numToChar** of (*factor*)
the **cos** of (*factor*)	the **round** of (*factor*)
the **exp** of (*factor*)	the **sin** of (*factor*)
the **exp1** of (*factor*)	the **sqrt** of (*factor*)
the **exp2** of (*factor*)	the **tan** of (*factor*)
the **ln** of (*factor*)	the **trunc** of (*factor*)
the **ln1** of (*factor*)	the **value** of (*factor*)

Properties

Background and Card Properties

name	ID	script
number		

Button Properties

autoHilite	rectangle	textHeight
icon	script	textSize
ID	showName	textStyle
location	style	visible
name	textAlign	
number	textFont	

Field Properties

autoTab	rectangle	textHeight
font	script	textSize
ID	scroll	textStyle
location	showLines	visible
lockText	style	wideMargins
name	textAlign	
number	textFont	
(*HyperCard* 2.0 only)		
dontSearch	dontWrap	fixedLineHeight

Global Properties

blindTyping	lockRecent	textArrows
cursor	lockScreen	userlevel
dragSpeed	lockScreen	visible
loc[ation]	powerKeys	
lockMessages	rect[angle]	

Painting Properties

brush	grid	multiSpace
centered	lineSize	polySides
filled	multiple	

Paint Text Properties

textAlign	textSize	textHeight
textFont	textStyle	

Script Properties

'name	freesize	size
script		

Special Symbols

--	to mark comments in code
&	concatenation
&&	concatenation with space
¬	continuation (Option-Return)

System Messages

arrowKey	functionKey	newField
closeBackground	help	newStack
closeCard	hide	openBackground
closeField	idle	openCard
closeStack	mouseDown	openField
controlKey	mouseEnter	openStack
deleteBackground	mouseLeave	quit
deleteButton	mouseStillDown	resume
deleteCard	mouseUp	returnKey
deleteField	mouseWithin	show
deleteStack	newBackground	startUp
doMenu	newButton	suspend
enterKey	newCard	tabKey

Other HyperTalk Words

Adjectives

all	long	short
abbreviated abbrev abbr		

Button Styles

transparent	rectangle	checkBox
opaque	roundRect	radioButton

Chunks

character char	list	word
item		

Chunk Types

characters chars	lists	words
items		

Constants

after	nine	space
down	one	ten
eight	pi	three
empty	quote	true
false	return	two
five	seven	up
four	six	zero

Containers

it	[the] message [window]
[the] message [box]	[the] msg [window]
[the] msg [box]	

Cursors

busy	I-beam	watch
crossbar	thick crossbar	

Effectnames (used with visual)

barn door close	scroll down	wipe right
barn door open	scroll left	wipe up
checkerboard	scroll right	zoom close
dissolve	scroll up	zoom in
iris close	venetian blinds	zoom open
iris open	wipe down	zoom out
plain	wipe left	
HyperCard 2.0 includes		
cut	stretch from bottom	
shrink to bottom	stretch from center	
shrink to center	stretch from top	
shrink to top		

Field Styles

opaque	scrolling	transparent
rectangle	shadow	

Image (used with visual)

black	card	white
inverse	gray	

Note Durations

w h q e s t x with . or 3

Notes

a b c d e f g with # or b
r (for rest)

Objects

background	card	stack
button	field	

Object Descriptors

me	previous	this
next	recent	

Object Modifiers

next	prev[ious]

Object Types

backgrounds	cards	fields
buttons		

Ordinal Constants

first	fifth	ninth
second	sixth	tenth
third	seventh	
fourth	eighth	

Prepositions

after	before	into

Special Ordinals

last	mid[dle]

Speed (used with visual)

fast	very fast	very slow[ly]
slow[ly]		

Text Operators

&&	contains	is not in
&	is in	

Window Names

card window	[the] message [window] [the] msg [window]
tool window	[the] message [box] [the] msg [box]
pattern window	

Appendix 3. Alphabetical Listing of HyperTalk Words

abbreviated (or abbrev or abbr) (adjectives)

abs (function)

after (constants)

after (preposition)

all (adjectives)

answer (command)

any (special ordinal)

arrowKey (system message)

ask (command)

atan (function)

autoHilite (property: button)

autoTab (property: field)

average (function)

background (object)

backgrounds (object type)

barn door close (effect name)

barn door open (effect name)

beep (command)

before (preposition)

black (image)

blindTyping (property: global)

brush (property: painting)

busy (cursor name)

button (object)

buttons (object type)

card (image)

card (object)

card window (window name)

cards (object type)

centered (property: painting)

character (or char) (chunks)

characters (or chars) (chunk types)

charToNum (function)

checkBox (button styles)

checkerboard (effect name)

choose (command)

click (command)

clicktext

closeBackground (system message)

closeCard (system message)

closeField (system message)

closeStack (system message)

contains (text operators)

controlKey (system message)

cos (function)

crossbar (cursor name)

cursor (property: global)

cut (effect name)

date (function)

delete (command)

deleteBackground (system message)

deleteButton (system message)

deleteCard (system message)

deleteField (system message)

deleteStack (system message)

dissolve (effect name)

doMenu (command)

doMenu (system message)

dontSearch (property: field)

dontWrap (property: field)

down (constants)

drag (command)
dragSpeed (property: global)
eight (constants)
eighth (ordinal constant)
empty (constants)
end (keyword)
enterKey (system message)
exit (keyword)
exp (function)
exp1 (function)
exp2 (function)
false (constants)
fast (speed)
field (object)
fields (object type)
fifth (ordinal constant)
filled (property: painting)
find (command)
first (ordinal constant)
five (constants)
fixedLineHeight (property: field)
font (property: field)
foundChunk (function)
foundField (function)
foundLine (function)
foundText (function)
four (constants)
fourth (ordinal constant)
freesize (property: script)
functionKey (system message)
get (command)
global (keyword)
go (command)
gray (image)
grid (property: painting)
group
help (system message)
hide *(command)*
hide (system message)
hide groups
I-beam (cursor name)
icon (property: button)
ID (property: background, card)
ID (property: button)
ID (property: field)
idle (system message)
if (keyword)
into (preposition)
inverse (image)
iris close (effect name)
iris open (effect name)
is in (text operators)
is not in (text operators)
it (container)
item (chunks)
items (chunk types)
last (special ordinal)
lines (chunk types)
lineSize (property: painting)
list (chunks)
ln (function)
ln1 (function)
location (property: button)
location (property: field)
lockMessages (property: global)
lockRecent (property: global)
lockScreen (property: global)
lockScreen (property: global)
lockText (property: field)
loc[ation] (property: global)
log2 (function)
long (adjectives)
max (function)
me (object descriptor)

message box (container)
message window (window name)
mid[dle] (special ordinal)
min (function)
mouse (function)
mouseClick (function)
mouseDown (system message)
mouseEnter (system message)
mouseH (function)
mouseLeave (system message)
mouseStillDown (system message)
mouseUp (system message)
mouseV (function)
mouseWithin (system message)
multiple (property: painting)
multiSpace (property: painting)
name (property: background, card)
name (property: button)
name (property: field)
name (property: script)
newBackground (system message)
newButton (system message)
newCard (system message)
newField (system message)
newStack (system message)
next (object descriptor)
next (object modifier)
nine (constants)
ninth (ordinal constant)
number (function)
number (property: background, card)
number (property: button)
number (property: field)
numToChar (function)
*offset (function)
on (keyword)
one (constants)
opaque (button styles)
opaque (field style)
openBackground (system message)
openCard (system message)
openField (system message)
openStack (system message)
pattern window (window name)
pi (constants)
pi (constants)
plain (effect name)
play (command)
polySides (property: painting)
powerKeys (property: global)
previous (object descriptor)
prev[ious] (object modifier)
print (command)
put (command)
quit (system message)
quote (constants)
quote (constants)
radioButton (button styles)
random (function)
recent (object descriptor)
rectangle (button styles)
rectangle (field style)
rectangle(property: button)
rectangle(property: field)
rect[angle] (property: global)
repeat (keyword)
reset (command)
resume (system message)
return (constants)
return (constants)
return (constants)
returnKey (system message)

round (function)

roundRect (button styles)

script (property: background, card)

script (property: button)

script (property: field)

script (property: script)

scroll (property: field)

scroll down (effect name)

scroll left (effect name)

scroll right (effect name)

scroll up (effect name)

scrolling (field style)

second (ordinal constant)

seconds (function)

select (command)

selectedChunk (function)

selectedField (function)

selectedLine (function)

selectedText (function)

send (keyword)

set (command)

seven (constants)

seventh (ordinal constant)

shadow (field style)

short (adjectives)

short (adjectives)

show (command)

show (system message)

show groups

showLines (property: field)

showName (property: button)

shrink to bottom (effect name)

shrink to center (effect name)

shrink to top (effect name)

sin (function)

six (constants)

sixth (ordinal constant)

size (property: script)

slow[ly] (speed)

space (constants)

space (constants)

sqrt (function)

stack (object)

startUp (system message)

stretch from bottom (effect name)

stretch from center (effect name)

stretch from top (effect name)

style (property: button)

style (property: field)

suspend (system message)

tabKey (system message)

tan (function)

ten (constants)

tenth (ordinal constant)

textAlign (property: button)

textAlign (property: field)

textAlign (property: paint text)

textArrows (property: global)

textFont (property: button)

textFont (property: field)

textFont (property: paint text)

textHeight (property: button)

textHeight (property: field)

textHeight (property: paint text)

textSize (property: button)

textSize (property: field)

textSize (property: paint text)

textStyle (property: button)

textStyle (property: field)

textStyle (property: paint text)

thick crossbar (cursor name)
third (ordinal constant)
this (object descriptor)
three (constants)
ticks (function)
time (function)
tool window (window name)
transparent (button styles)
transparent (field style)
true (constants)
trunc (function)
two (constants)
up (constants)
userlevel (property: global)
value (function)
venetian blinds (effect name)
very fast (speed)
very slow[ly] (speed)
visible (property: button)
visible (property: field)
visible (property: global)
visual (command)
wait (command)
watch (cursor name)
white (image)
wideMargins (property: field)
wipe down (effect name)
wipe left (effect name)
wipe right (effect name)
wipe up (effect name)
word (chunks)
words (chunk types)
zero (constants)
zoom close (effect name)
zoom in (effect name)
zoom open (effect name)
zoom out (effect name)

Index

⌘ 59
& 59, 138
  as text operator 209
&& 190
  as text operator 209
-- 139, 141
¬ 138
Abbreviated 42
  name of button 62
  with field name 80
Abs 237
Accidental
  in music 73
After
  with put 207
All
  with show 41
Animation 173
  flipping cards 173
  using the Tool window 175
Answer
  syntax of 215
Any 147
Ask
  syntax 216
Atan 237
AutoHilite
  check box 25
  problems with 69
  property of buttons 63
  with hilite 69
AutoTab
  property of fields 80
Average 235
Background
  description 6
  editing 14
  getting blank 11
  keeping default 11
Background property
  ID 193
  name 193
  number 193
  script 193
Backgrounds
  In stack design 119
  scripting 193
Bakus-Naur notation 94
Barn door
  using 118
Beep 28
  syntax of 28
Beep sound 73
Before
  with put 207
BlindTyping
  property 106
Browse 152
Browse tool
  defined 14
Brush 152, 156
Bucket 152
Bucket tool
  using 14
Busy cursor
  using 248
Button
  copying 20
  getting script of 54
Button property
  autoHilite 63
  icon 63
  ID 63
  location 66
  name 62, 63
  rectangle 67
  script 64
  showName 63
  Style 65
  textAlign 131
  textFont 131
  textHeight 131
  textSize 131
  textStyle 131
  visible 127
Button style
  hilite 66
Button tool 152

Buttons
as components of stacks 7
hilited form 69
invisible 70
level of 134
linking 17, 26
lost 70
new 17
off the card 70
problems referencing 58
properties of 61
scripting 25
shortcut for copying 52
unhilited form 69
Capitalization
in HyperTalk words 28
Card
copying 16
new 12
Card property
ID 193
name 193
number 193
script 193
Card size
in HyperCard 2.0 40
Cards
flipping through 41
scripting 193
Case statement 228
Centered 157
Character
as a chunk 202
CharToNum 235
CheckBox
button style 65
Checkerboard
using 118
Choose 151
Chunk 202
character 202
item 202
line 202
word 202
Chunk type 56
Click
syntax of 152
CloseField
problems with 128
Command key 59
Commandkey 153
Commands 43
and functions 89
Comments
in scripting 139
Comments symbol 139
for debugging 141
Concatenation symbol 59, 138
Concatenation with space 190
Condition 223
Conditionals 223
Constants
in HyperTalk 95
numeric 146
special ordinal 147
with text 208
Container
Message box 36
selection 211
Containers 48, 239
Contains
as text operator 209
Continuation symbol 138
Conventions
handler names 28
in stack design 123
Coordinates 66
Coordinates of pointer
finding 70
Copying
button 20
Copying buttons
shortcut 52
Copying fields
shortcut 52
Cos 237
Cursor
property 107
Cursors
standard 107
Curve 152
Date
syntax of 42
Delete 206
Descriptors
object 143
Display stacks 117
Dissolve
using 117

DoMenu
syntax 181
DontSearch
property of fields 81
DontWrap
property of fields 81
Dotted notes 74
Drag
syntax of 153
using 155
DragSpeed 155
Duration
symbols to use with Play 74
Else
in If 225
Empty 50, 95
End 27
Eraser 152
Exp 237
Exp1 237
Exp2 237
Expression 43
False 63
Field
creating 20
dontSearch 81
getting script of 54
pop up 127
text in 21
Field property
autoTab 80
dontWrap 81
fixedLineHeight 81
font 129
ID 80
lockText 80
name 80
Number 80
showLines 80
Style 84
textAlign 130
textFont 130
textHeight 130
textSize 130
textStyle 130
wideMargins 80
Field tool 152
Fields
as components of stacks 7
as containers 48, 239
as objects 48
level of 134
locked 86
maximum size of contents 239
problems referencing 58
properties of 79
putting text into 48
removing text from 50
scripting 51
shortcut for copying 52
using 47
Filled 157
Filter stacks 117
Find
syntax of 204
FixedLineHeight
property of fields 81
Flipping cards
to animate 173
Font
property of fields 129
Formatting
as an aid to debugging 232
FoundChunk 205
FoundField 205
FoundLine 205
FoundText 205
Freesize
stack property 195
Function 27, 43
Functions
general form 235
in scripts 43
introduction 41
model of 90
Generic names 143
Get
syntax 243
Getting started 9
Global
properties 105
Global command 244

Global properties
blindTyping 106
cursor 107
lockMessages 108
lockRecent 107
lockScreen 108
powerKeys 107
textArrows 107
Go
complete syntax 91
syntax 29
Grammar
of HyperTalk 89
Grid 158
Handler
defined 27
structure of 27
Handlers
defined 100
order of 100
writing 137
Hardware needed 2
Helper word 31
Hide
menu bar 40
syntax 38
Hilite
button style 66
problems with 69
with autohilite 69
Home Card 9
HyperCard
as an object-like programming environment 9
definition of 5
uses of 5
version of 2
Hypermedia 1
HyperTalk 1
as a programming language 28
Hypertext 1
Icon
property of buttons 63
property when no icon 63
Icons
conventions for use 120
in stack design 120
ID
background property 193
card property 193
property of buttons 63
property of fields 80
Idle 231
message 230
If then 223
If then else 225
In
in object descriptors 143
with put 207
Interactive programming 215
Into
with put 69
Iris
using 118
Is in
as text operator 209
Is not
as text operator 209
It
as a container 239
as local variable 221
cannot be global 245
scope of 243
with answer 216
with Ask 217
Item
as a chunk 202
Jump linear stacks 113
Keywords
in HyperTalk 100
Language element 29
Lasso 152
Last 147
Length
with chunk expressions 203
Level of
buttons and fields 134
Line 152
as a chunk 202
in fields 86
Linear stacks 113
LineSize 158
Linking
buttons 26

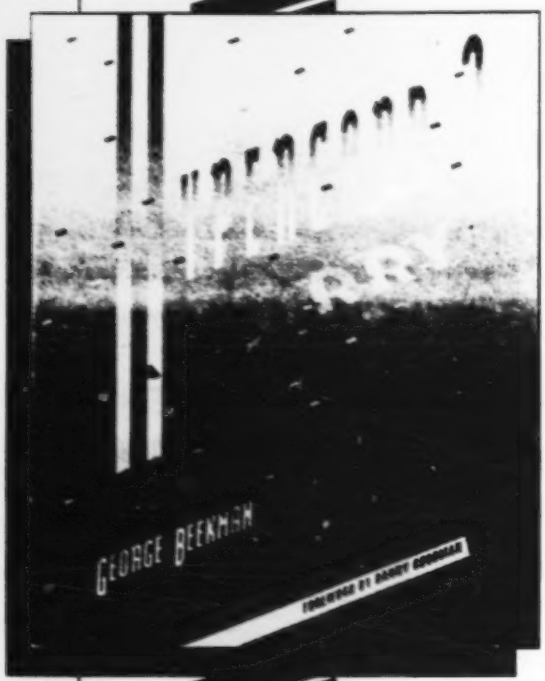
Literals
In HyperTalk 95
Ln 237
Ln1 237
Local variable
it 221
Location property 66
of windows 109
LockMessages
property 108
LockRecent
property 107
LockScreen
property 108
LockText
property of fields 80
Long 42
name of button 62
with Field name 80
Long lines
in music 138
Looping 247
Math functions 235
Max 235
Me 144
MenuBar 40
Message box
as a container 239
as container 36
for debugging and texting 39
introduction to 35
moving 36
other names for 38
to aid in debugging 44
typing in 37
using with fields 47
Message handler 27
Message passing
problems with 101
using send 187
Message passing hierarchy 96
Message Watcher window 96
Messages
in HyperTalk 95
passing 27
sent to buttons 99
sent to cards 99
sent to fields 99
Middle 147
Min 235
Modularizing scripts 137
Mouse 253
MouseClick 253
MouseH 145
Mouseloc 70
MouseV 145
MouseWithin 27
MuitiSpace 157
Multiple 157
Multiple if statements 228
Music
chart of notes 74
stopping 76
Name
background property 193
button property 62
card property 193
property of fields 80
stack property 195
Network stacks 116
New background
creating 16
Next 36, 143
Next and previous
with backgrounds 144
Notes
chart of 74
dotted 74
how represented 73
triplets 74
Number 55
background property 193
card property 193
property of buttons 63
property of fields 80
syntax of 93
Numbers
as text 213
NumToChar 235
Object descriptors 143
special 143
Object oriented programming 9
Object-oriented language 94
Objects
in HyperCard 8, 94
Octave
setting 74
Of
in object descriptors 143

Offset 203
On 27
Opaque
button style 65
with fields 85
OpenBackground message 197
OpenCard
problems with 198
OpenStack message 197
Optional element 31
Optional parameter 29
Optional phrases 92
OptionKey 153
using with drag 178
Order of
handlers in the script editor 100
Oval 152
Oval tool
using 14
Paint text 13
and text in fields 160
properties of 159
Painting properties 156
Parameter 29
optional 29
Parameters 91
multiple 259
scope of 255
using 255
Pattern 158
Pattern window 37, 38
Patterns
Selecting 158
Pencil 152
Pixels 66
Play
problems with 76
syntax 73
Point and click stack 117
Pointer
arrow 35
I-beam 35
pointing hand 35
vertical bar 35
Pointers
standard 107
Polygon 152
PolySides 158
Pop up field 127
PowerKeys
property 107
Prepositions
in object descriptors 143
problems with 69
Prerequisites 1
Previous 143
Previous and next
with backgrounds 144
Print Card 168
from File menu 168
size of printed cards 169
Print Report
from the File menu 167
Print Report dialog box 167
Print Stack
from File menu 165
Print Stack dialog box 165
Printing 165
in the script editor 169
part of a script 171
Priority of objects
changing 135
Properties
field 79
global 105
In HyperTalk 94
in Objects menu 61
introduction to 61
painting 156
Put
syntax of 43
to check property values 101
with fields 207
Quotation marks
in scripts 27
use of 62
Quote
as a constant 208
RadioButton
button style 65
Random 182
Recent 32, 144
Rectangle 152
button style 65
property of buttons 67
with fields 85
Rectangle property
of windows 109

Rectangle tool
using 14, 16
Regular polygon 152
Repeat 174, 247
Repeat for 175, 247
Repeat until 251
Repeat while 251
Repeat with 248
Reset paint 159
Rest
in music 73
Return
as a constant 208
Return symbol 51
Round 236
Round rectangle 152
RoundRect
button style 65
Scope
of variables 260
Scope of variables
problems with 245
Screen size
standard 40
Script
background property 193
card property 193
of Button
getting 54
of field
getting 54
property of buttons 64
stack property 195
Script editor
printing from 169
Scripting
backgrounds 193
buttons 25
cards 193
defined 100
sields 51
stacks 193
Scripts
modularizing 137
stopping 198
where to place 197
Scroll
property 86
Scrolling
value of style of fields 85
with fields 85
Seconds 42
Select 152
syntax of 210
SelectedChunk 211
SelectedField 211
SelectedLine 211
SelectedText 211
Selection
as a container 239
container 211
Send
introduction to 140
syntax 187
to pass messages using scripts 187
to test handlers 140
Set
properties of buttons 63
Shadow
with fields 85
ShiftKey 153
Short 42
name of button 62
with field name 80
Show
menu bar 40
syntax 38
ShowLines
property of fields 80
ShowName
property of buttons 63
Shrink
using 118
Sin 237
Single frame stacks 117
Size
limits of 195
stack property 195
Sound
beep 73
boing 73
flute 73
harpsichord 73
Sounds
available in HyperCard 73
digitized 73
setting 73

Space
as a constant 208
Special object descriptors 143
Spray can 152
Sqrt 236
Stack
new 11
point and click 117
structure of 113
Stack design
conventions 123
Stack property
freesize 195
name 195
script 195
size 195
Stacks
designing 113
display 117
filter 117
jump linear 113
linear 113
network 116
scripting 193
single frame 117
tree structure 114
Statement
HyperTalk 27
Stop
with Play 76
Stopping
script when running 198
Stopping music 76
Stretch
using 118
Style
property of buttons 65
property of fields 84
Styles
of buttons 65
Syntax 28
Syntax diagrams
explanation of 91
Tan 237
Tempo
with Play 75
Text 152
in fields 21, 201
paint 13, 21
properties of 159
paint versus text in fields 22
putting into fields 48
removing 50
Text in fields
and paint text 160
text operator
& 209
&& 209
contains 209
is in 209
is not 209
Text tool
using 13
TextAlign
property of buttons 131
property of fields 130
TextAline
with paint text 159
textArrows
property 107
TextFont
property of buttons 131
property of fields 130
with paint text 159
TextHeight
property of buttons 131
property of fields 130
with paint text 159
TextSize
property of buttons 131
property of fields 130
with paint text 159
TextStyle
property of buttons 131
property of fields 130
The
rules for using 148
use with functions 42
use with go 36
with properties 62
Then
in If 223
This 144
Ticks 42
Time 42

To
in chunk expressions 203
with set 69
Tool names 152
Tool window
tearing off 13
use with HyperTalk 151
Tools
using 14
using in HyperTalk 151
Tools window
using 12
Transparent
button style 65
with fields 85
Tree structure
of stacks 114
Triplets 74
True 63
Trunc 236
User level
setting 10
Userlevel
property 105
setting 105
Value 236
Values
in HyperTalk 95
Variables
restrictions on names of 239
scope of 239
Venetian blinds
using 118
Version of HyperCard 2
Visible
property of buttons 127
Visible property
of windows 108
Visual
command 30
effect names 31
image 31
speed 31
syntax 30
Visual effect
barn door
using 118
checkerboard
using 118
dissolve
using 117
iris
using 118
shrink
using 118
stretch
using 118
wipe
using 118
zoom
using 118
Visual effects 29
combination 118
in stack design 117
on color monitors 30
problems with 32
using 25
Voices
in HyperCard 73
Wait
syntax 34
Wait until 252
Wait while 252
WideMargins
property of fields 80
Window
Pattern 37
Tool 151
Windows
card as 37
Message box as 37
properties of 108
Wipe
using 118
Word
as a chunk 202
Zoom
using 118



Making Tracks Through Stacks

After a few hours with George Beekman's book, even novice computer users will be able to design, build, and customize visually impressive and conceptually complex stacks to perform a variety of tasks.

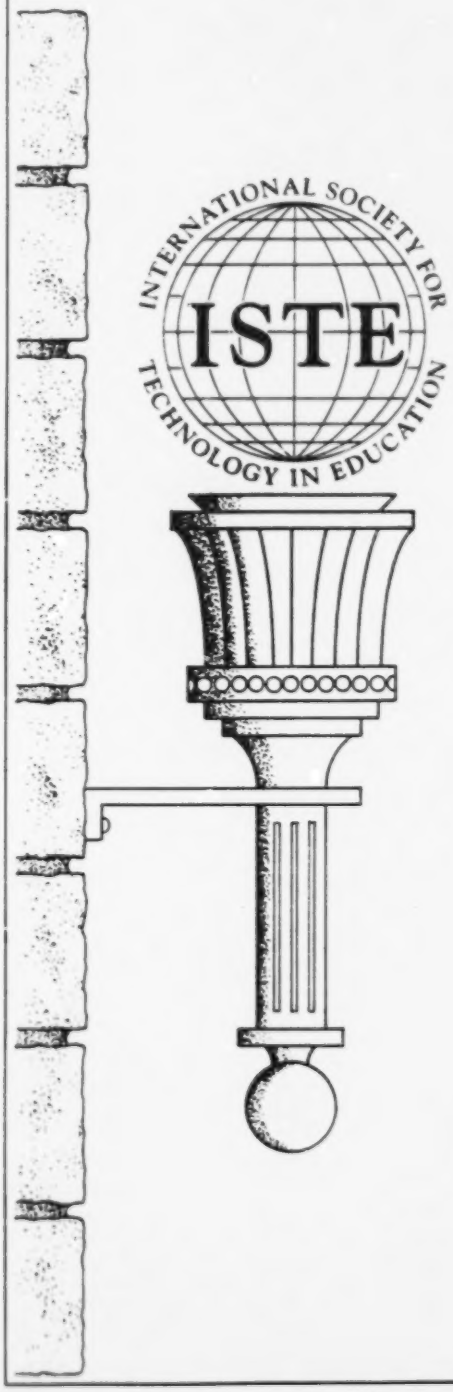
HyperCard In A Hurry systematically presents the fundamentals of *HyperCard*® through a series of self-study, hands-on sessions. The first two sessions introduce *HyperCard* as a tool for information storage and retrieval. The four remaining sessions illustrate essential techniques and tricks for building a variety of *HyperCard* stacks.

Available as **HyperCard In A Hurry,** version 1.0 or **HyperCard In a Hurry 2**, version 2.0.

ISTE
International Society for Technology in Education
1787 Agate Street, Eugene, OR 97403-1923
Order Desk: 800/336-5191 Fax: 503/346-5890

BEST COPY AVAILABLE

The International Society for Technology in Education Touches All Corners of the World



As the largest international non-profit professional organization serving computer using educators, we are dedicated to the improvement of education through the use and integration of technology.

Drawing from the resources of committed professionals worldwide, ISTE provides information that is up-to-date, compelling, and relevant to your educational responsibilities.

Rely on ISTE support for:

- Periodicals
- Books and Courseware
- Special Interest Groups
- Independent Study Courses
- Professional Committees
- Private Sector Council

Be part of the international sharing of educational ideas and technology. Join ISTE today.

International Society for Technology in Education
1787 Agate Street, Eugene, OR 97405-1923
Order Desk: 800/336-5191 Phone: 503/346-4414
Fax: 503/346-5891 CompuServe: 70014,2117 Bitnet: ISTE@Oregon
Internet: ISTE@Oregon.uoregon.edu ISTE.Net: iste.office

BEST COPY AVAILABLE